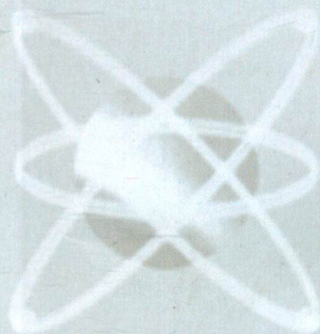


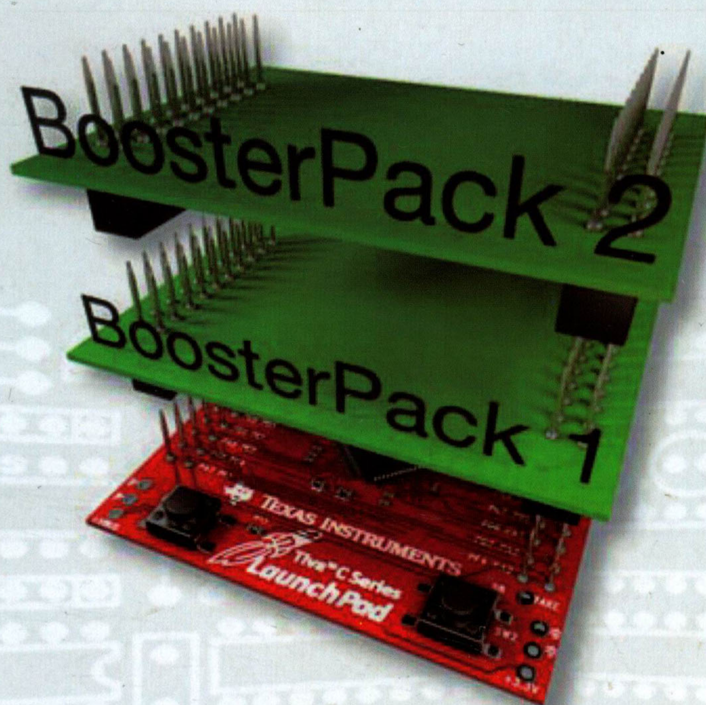
Technology by TEXAS INSTRUMENTS 中国大学计划教材

电气信息工程丛书

基于固件的ARM Cortex M4原理及应用

刘杰 陈昌川 编著







- 配有各章程序及相关资源，下载网址为www.cmpbook.com
- 采用真实硬件：EK-TM4C123GXL和虚拟硬件：Proteus 8.1



机械工业出版社
CHINA MACHINE PRESS



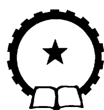
VISIT…

LANZAROTE Caliente.COM

电气信息工程丛书

基于固件的 ARM Cortex M4 原理及应用

刘 杰 陈昌川 编著



机 械 工 业 出 版 社

本书围绕 TI TM4C123G 的固件库函数这一主线，介绍了 TM4C123G6HPM 微处理器的基本外设特点、结构与功能，固件库的函数功能及其使用。本书采用了真实硬件 EK－TM4C123GXL LaunchPad 实验板（包括 DK－TM4C123G）与虚拟硬件 Proteus 8.1 相结合的方式来介绍基于固件的软件编程与测试方法，以利于有真实板卡但资源不足或无 EK－TM4C123GXL 板卡的读者学习与测试基于固件的代码之用。

本书可供嵌入式工程师在基于固件的 ARM Cortex M4 开发时查阅，也可作为高校电类专业学习 ARM Cortex M4 的入门教材。

图书在版编目（CIP）数据

基于固件的 ARM Cortex M4 原理及应用/刘杰，陈昌川编著．—北京：机械工业出版社，2015.9

（电气信息工程丛书）

ISBN 978-7-111-51624-8

Ⅰ．①基…　Ⅱ．①刘…②陈…　Ⅲ．①微处理器　Ⅳ．①TP332

中国版本图书馆 CIP 数据核字（2015）第 226920 号

机械工业出版社（北京市百万庄大街 22 号　邮政编码 100037）

责任编辑：时　静

责任校对：张艳霞

责任印制：李　洋

北京机工印刷厂印刷（三河市南杨庄国丰装订厂装订）

2015 年 10 月第 1 版 · 第 1 次印刷1

84mm×260mm · 32 印张 · 792 千字

0001－3000 册

标准书号：ISBN 978-7-111-51624-8

定价：89.90元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务

服务咨询热线：010-88361066

读书购书热线：010-68326294

010-88379203

网络服务

机工官网：www.cmpbook.com

机工官博：weibo.com/cmp1952

金书网：www.golden-book.com

教育服务网：www.cmpedu.com

封面无防伪标均为盗版

前　言

TI Tiva™ C 系列微控制器采用基于 ARM® Cortex™ – M4 内核的卓越架构，具备强大的集成能力，提供了成熟的软件和开发工具生态系统，是嵌入式工程师的理想选择。为了提供最佳的性能和灵活性，Tiva™ C 系列架构推出了具有 FPU、各种集成存储器以及可编程 GPIO 的 80 MHz Cortex – M4 微控制器。Tiva™ C 系列可集成适合特殊应用的外设，以及广泛的软件工具选项，能大大降低电路板成本，缩短产品的设计周期，是用户理想的成本效益型解决方案。

随着 TI Stellaries M3 系列微控制器逐步退出历史舞台，Tiva™ C 系列芯片可帮助用户缩短产品上市时间、节省开发成本，成为高性能 32 位应用的优选。但市面上或网络中有关 TM4C123G 系列微控制器的书籍和资料还很少，使初学者很难上手，所以编撰一本有关 Tiva™ C 系列微控制器的技术手册和入门级教科书很有必要，本书就是在这样的形势下实施的。

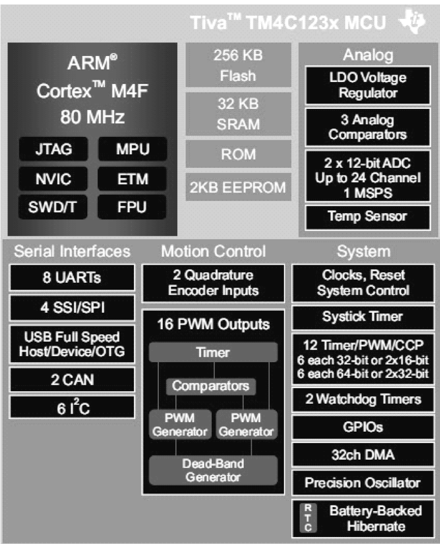
本书历时一年多，于 2014 年 3 月底完成初稿，因笔者 4 月份生病住院而耽误了书的进展。今年 1 月按照新版固件库（SW – TM4C – DRL – UG – 2. 0. 1. 11577）重新对原有固件库函数进行了增减与修订，修改了各章的外设功能简介，增加了基于 Proteus 虚拟硬件测试的例程，并于 2015 年 4 月 12 日完成全部工作。

对于初学者来说，模仿不失为一种快速掌握 Tiva™ C 系列微控器软件编程的有效方法，参考文献中提到的一家公司为 Stellaries M3 能被国内的广大用户接受做了大量的推广工作，并留下了许多宝贵的网络资源。国内的其他科技公司和大学也为 Stellaries M3 的推广做了一些有益工作，但它们采用了不同的 ARM Cortex 开发软件，这给初学者使用这些宝贵资源带来了不少的麻烦。为了规避这些问题，本书较详细地介绍了 Keil5 for ARM、IAR for ARM、CCS 6 软件的使用方法，为没有 LaunchPad 实验板或硬件、测试仪器匮乏的读者专门介绍了基于 Proteus 8. 1 的软件测试方法，因为仅就软件编程与测试方法来说，ARM Cortex M3 和 M4 并无多大区别。

本书摒弃了传统的通过配置寄存器来开发 AMR Cortex M4 的程序开发模式（51 单片机式的），可大大加快编写软件与测试的进度，降低开发者的入门门槛。因为目前国外多采用基于模型设计的软件开发方法，仅需开发者把主要精力投入算法模型的研究上，代码由计算机自动生成。基于固件的开发本质上也继承了基于模型的观点，让工程师从详细了解超 1000 页的 M4 芯片数据手册，以及配置寄存器代码的长时间劳动中解放出来。把这些困难与繁重的工作交由芯片生产厂家去完成（编写各寄存器的固件库函数代码），开发者仅需专注项目的实现方法即可。本书紧扣固件库函数这一核心，共 16 章，详细介绍了固件库函数的功能及使用方法，在书后附录中给出了常用固件库函数的功能简介（为了压缩本书的厚度省略了注意事项，读者在查阅附录时请留意，如果遇到使用问题可参考 SW – TM4C – DRL –

UG－2.0.1.11577 固件库原文)，以期初学者能快速掌握基于固件的 ARM Cortex M4 的软件开发与测试方法。

有关 ARM Cortex 架构的书籍及网络资源很多，本书不再赘述这部分内容。对于不了解 ARM Cortex 架构的读者，在学习本书之前请先行阅读 ARM Cortex－M3 权威指南一书，这里仅给出 Tiva™ TM4C123x 的模块框图，如下图所示。



本书由刘杰、陈昌川编写，写作过程中得到了美国德州仪器中国大学部谢胜祥工程师、大学部经理沈洁、黄争、潘亚涛等大力支持，他们提供了本书全部的实验板卡、资料等。程泳、郭丹、李晗、吴仪炳、陈添丁、杨元廷、史进、谢文福、杨叶腾、陈松雷、寿永勇、余延臻、林东灿、林亮亮、许惠敏 、王爱忠、苏泓、史永祥、陈鸿霖、周楠、赵建欣、王丽琴、谭笑、林静、黄荣、高建鸿、杜程远、张志鸿、张伟敏、吴承清、林肖、李加滨、江丽珍、黄冠莉、陈阳、董晓芳、陈志成、姜杨、彭浩书等同学参加了个别章节的结构和固件库函数的初始翻译与资料整理工作，机械工业出版社的时静编辑对本书的进展进行了全程监控并对书的结构提出了宝贵的建议，在此一并感谢，没有他们的辛勤劳动与帮助不可能完成本书的编撰工作。

由于书中 TI 技术资料的翻译与整理工作量较大，而且时间紧任务重，加之自己的水平有限，肯定存在对 TI 技术文献的理解歧义和错误，敬请读者批评指正。

刘　杰

2015.4 于福大怡园

目　　录

前言
第 1 章　开发工具使用入门 …… 1
1.1　下载与安装所需的软件 …… 1
1.2　第一个基于 CCS6 的 hello world 工程 …… 2
1.2.1　导入已存在的工程 …… 2
1.2.2　创建一个新工程 …… 6
1.2.3　LM 闪存编程器 …… 9
1.3　Keil for ARM 入门基础 …… 12
1.3.1　导入一个 hello 工程 …… 12
1.3.2　创建一个 hello 工程 …… 15
1.4　IAR Embedded Workbench for ARM 入门基础 …… 19
1.4.1　打开一个现有工程 …… 19
1.4.2　创建一个新工程 …… 21
第 2 章　EK-TM4C123GXL 及 Proteus 简介 …… 26
2.1　EK-TM4C123GXL 简介 …… 26
2.1.1　TM4C123GXL 的特点 …… 26
2.1.2　评估板模块框图 …… 27
2.2　Proteus 8.1 简介 …… 28
2.2.1　新增功能 …… 28
2.2.2　Proteus 8.1 界面简介 …… 28
2.2.3　如何寻找 Proteus 中的元器件 …… 29
2.2.4　虚拟仪器的使用 …… 30
2.2.5　基于 Proteus 8.1 的 M3 编程与测试 …… 33
2.2.6　基于 Proteus 8.1 的 M3 代码测试 …… 38
第 3 章　通用异步收发器模块（UART） …… 41
3.1　UART 模块 …… 41
3.1.1　UART 的特点 …… 41
3.1.2　UART 的结构框图 …… 42
3.1.3　信号描述 …… 42
3.1.4　UART 模块功能的简要介绍 …… 44
3.2　UART 固件库函数 …… 49
3.2.1　UART 固件库结构 …… 49
3.2.2　UART 的基本操作 …… 50
3.3　例程 …… 50
第 4 章　模数转换器（ADC） …… 57
4.1　ADC 模块 …… 57
4.1.1　ADC 特点 …… 57
4.1.2　ADC 模块框图 …… 58
4.1.3　信号描述 …… 60
4.1.4　功能简介 …… 60
4.2　ADC 固件库函数 …… 66
4.3　例程 …… 66
第 5 章　通用输入/输出（GPIO） …… 79
5.1　GPIO 模块 …… 79
5.1.1　GPIO 特点 …… 79
5.1.2　GPIO 模块框图 …… 80
5.1.3　功能简介 …… 80
5.1.4　寄存器映射及寄存器描述 …… 82
5.2　GPIO 固件库函数 …… 87
5.3　例程 …… 88
第 6 章　模拟比较器（COMP） …… 99
6.1　COMP 单元 …… 99
6.1.1　COMP 特点 …… 99
6.1.2　COMP 模块框图 …… 99
6.1.3　信号描述 …… 99
6.1.4　功能简介 …… 100
6.1.5　寄存器映射 …… 102
6.2　COMP 固件库函数 …… 102
6.3　例程 …… 103
第 7 章　系统定时与中断控制 …… 113
7.1　NVIC 模块 …… 114
7.1.1　NVIC 模块的特点 …… 114

7.1.2 功能描述 …………………………… 115
7.1.3 中断优先级 ………………………… 116
7.1.4 中断异常 …………………………… 116
7.1.5 寄存器映射 ………………………… 116
7.2 SysTick 与 NVIC 固件库函数 … 118
7.2.1 SysTick 固件库 …………………… 118
7.2.2 NVIC 固件库 ……………………… 119
7.3 例程 ………………………………… 119
第 8 章 内部集成电路接口（I2C） … 136
8.1 I2C 单元 …………………………… 136
8.1.1 I2C 特点 …………………………… 136
8.1.2 I2C 模块框图 ……………………… 137
8.1.3 信号描述 …………………………… 137
8.1.4 功能描述 …………………………… 138
8.2 I2C 固件库函数 …………………… 141
8.2.1 主机操作 …………………………… 141
8.2.2 从机操作 …………………………… 142
8.2.3 I2C 固件库描述 …………………… 143
8.3 例程 ………………………………… 143
8.3.1 主从回环例程 ……………………… 143
8.3.2 基于 I2C 的 EEPROM 读写例程 ………………………………… 151
第 9 章 同步串行接口（SSI） ……… 162
9.1 SSI 单元 …………………………… 162
9.1.1 SSI 的特点 ………………………… 162
9.1.2 模块框图 …………………………… 163
9.1.3 信号描述 …………………………… 164
9.1.4 功能简介 …………………………… 164
9.1.5 寄存器映射 ………………………… 171
9.2 SSI 固件库函数 …………………… 172
9.3 例程 ………………………………… 173
第 10 章 内部存储器 ……………… 183
10.1 内部存储器单元 ………………… 183
10.1.1 模块框图与控制逻辑 ………… 183
10.1.2 功能简介 ………………………… 183
10.2 闪存固件库函数 ………………… 190
10.3 使用 ROM ……………………… 191
10.3.1 直接 ROM 调用 ………………… 191
10.3.2 映射 ROM 调用 ………………… 192
10.3.3 ROM 固件更新 ………………… 193
10.4 EEPROM 固件库函数 ………… 194
10.5 例程 ……………………………… 195
10.5.1 写闪存例程 ……………………… 195
10.5.2 读写 EEPROM 例程 …………… 198
第 11 章 通用定时器（GPTM） …… 203
11.1 通用定时器单元 ………………… 203
11.1.1 主要特点 ………………………… 203
11.1.2 GPTM 模块框图 ……………… 204
11.1.3 信号描述 ………………………… 205
11.1.4 功能简介 ………………………… 205
11.2 GPTM 固件库函数 …………… 213
11.3 例程 ……………………………… 213
第 12 章 脉冲宽度调制（PWM） …… 224
12.1 PWM 单元 ……………………… 224
12.1.1 PWM 的主要特点 ……………… 224
12.1.2 PWM 的模块框图 ……………… 225
12.1.3 信号描述 ………………………… 225
12.1.4 功能简介 ………………………… 225
12.2 PWM 固件库函数 ……………… 231
12.3 例程 ……………………………… 231
第 13 章 微直接存储器访问（μDMA） ………………………… 238
13.1 μDMA 单元 …………………… 238
13.1.1 μDMA 的特点 ………………… 238
13.1.2 μDMA 模块框图 ……………… 239
13.1.3 功能简介 ………………………… 239
13.2 μDMA 固件库函数 …………… 247
13.3 例程 ……………………………… 247
第 14 章 通用串行总线控制器（USB） …………………………… 259
14.1 USB 简介 ……………………… 259
14.2 TM4C123GH6PM USB 控制器 … 266
14.2.1 USB 的特点 …………………… 266
14.2.2 USB 模块框图 ………………… 267
14.2.3 USB 信号描述 ………………… 267
14.2.4 USB 功能描述 ………………… 268
14.3 USB 固件库函数 ……………… 273
14.3.1 USB 的分层框架结构 ………… 273

14.3.2 Driverlib 库函数介绍 …………… 275
14.3.3 USBlib 库函数介绍 ……………… 279
14.4 例程 …………………………………… 283
第 15 章 FatFS 文件读取实验 ………… 300
15.1 SD 卡概述 …………………………… 300
15.1.1 SD 卡的内部结构及信号描述 … 300
15.1.2 SD 卡的命令 ………………………… 302
15.1.3 SD 卡的功能描述 ………………… 304
15.1.4 SD 卡驱动程序解读 ……………… 308
15.2 SD 卡 FatFS 文件读取实验 …… 320
15.2.1 FatFS 文件系统简介 ……………… 320
15.2.2 实验硬件连接图 ………………… 321
15.2.3 导入 sd_card 工程 ……………… 322
第 16 章 基本图形库（Grlib） ……… 335
16.1 图形库与液晶屏概述 …………… 335
16.1.1 图形库概述 ………………………… 335
16.1.2 液晶屏简介 ………………………… 336
16.2 TivaWare 图形库简介 …………… 346
16.2.1 图形库的特点 …………………… 346
16.2.2 图形库源代码 …………………… 347
16.2.3 图形固件库函数 ………………… 348
16.2.4 实用工具（Utilities） …………… 365
16.2.5 预定义的颜色参考 ……………… 367
16.3 例程 …………………………………… 368
附录 ………………………………………… 373
附录 A 第 3 章附录：UART 固件库函数简介 ………………………… 373
附录 B 第 4 章附录：ADC 固件库函数简介 ………………………… 385
附录 C 第 5 章附录：GPIO 固件库函数简介 ………………………… 397
附录 D 第 6 章附录：模拟比较器固件库函数简介 ……………… 409
附录 E 第 7 章附录：SysTick 与 NVIC 固件库函数简介 ……………… 412
E.1 SysTick 固件库函数 ………………… 412
E.2 NVIC 固件库函数 …………………… 413
附录 F 第 8 章附录：I2C 固件库函数简介 ………………………… 417
附录 G 第 9 章附录：SSI 固件库函数简介 ………………………… 429
附录 H 第 10 章附录：内部存储器的固件库函数简介 ……………… 435
H.1 闪存（Flash）固件库函数 ……… 435
H.2 闪存保护单元（MPU）固件库函数 ……………………………………… 438
H.3 EEPROM 固件库函数 ……………… 441
附录 I 第 11 章附录：GPTM 固件库函数简介 ………………………… 448
附录 J 第 12 章附录：PWM 固件库函数简介 ………………………… 458
附录 K 第 13 章附录：μDMA 固件库函数简介 ………………………… 473
附录 L 第 14 章附录：USB DriverLib 固件库函数简介 ……………… 481
参考文献 ………………………………… 503

第 1 章

开发工具使用入门

本章将简单介绍 CCS6、Keil for ARM、IAR for ARM 与 SW－EK－TM4C123GXL 驱动程序的安装及其使用方法。

本章的主要内容：

- ➢ CCS5 的安装及使用方法
- ➢ Keil for ARM 的安装及使用方法
- ➢ IAR for ARM 的安装及使用方法
- ➢ ICDI 驱动程序的安装



1.1 下载与安装所需的软件

SW－EK－TM4C123GXL 软件包的下载地址：http://www.ti.com/tool/SW－EK－TM4C123GXL

主要包括以下软件：

① 基于 CCS5/6 的软件包。

② 基于 IAR 的软件包。

③ 基于 KEIL 的软件包。

待下载的三个软件包如图 1-1 所示，已下载的 EK－TM4C123GXL－CCS 软件包内容如图 1-2 所示，然后按照手册的说明安装这三个软件包。限于篇幅这里不再详述。

Part Number	Buy from Texas Instruments
EK-TM4C123GXL-CCS: TivaWare for C Series and Code Composer Studio for the Tiva C Series TM4C123G LaunchPad	Free Get Software
EK-TM4C123GXL-IAR: TivaWare for C Series and IAR Embedded Workbench for the Tiva C Series TM4C123G LaunchPad	Free Get Software
EK-TM4C123GXL-KEIL: TivaWare for C Series and Keil RVMDK for the Tiva C Series TM4C123G LaunchPad	Free Get Software

图 1-1　待下载的软件

注意：三个软件包与 LaunchPad 板子相关的驱动程序都是相同的，只安装一次即可。

双击菜单 TivaWare 中的 SW－EK－TM4C－2.0.1.11577.exe，开始安装 TivaWare 开发软件包（如图 1-2 所示），安装完成后的开发软件包目录树如图 1-3 所示。

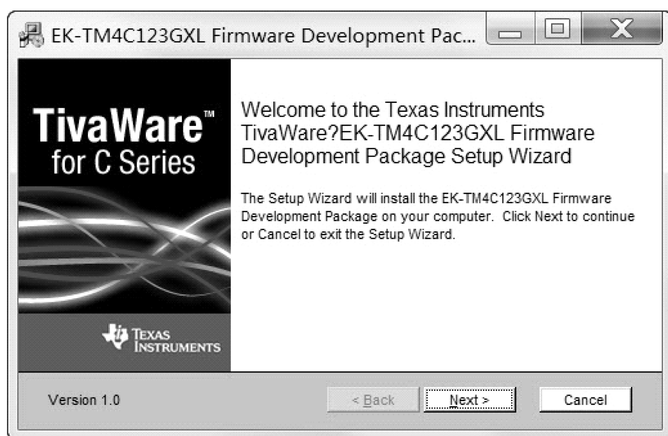


图 1-2　安装 EK－TM4C123GXL 开发包



图 1-3　TivaWare 开发软件包目录



1.2　第一个基于 CCS6 的 hello world 工程

1.2.1　导入已存在的工程

1）按默认状态打开一个工作空间，如图 1-4 所示。

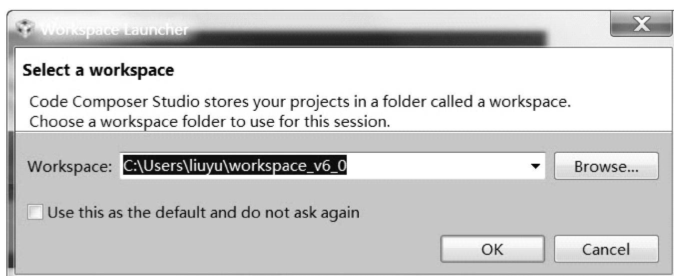


图 1-4　打开的默认工作空间

2）导入库函数。在主菜单栏的 Project 下拉菜单中选择 Import Existing CCS Eclipse Project 子菜单，在弹出的对话框中按 Browser 按钮寻找并导入 driverlib（驱动库）、usblib（usb 库）和 grlib（图形库）库函数，同时选中 Copy project into workspace 选项，如图 1-5、1-6 所示。

注意：若用户安装的是 SW－EK－TM4C－2.0.1.11577 软件包，则图 1-6 中的 TivaWare_C_Series 目录应该替换成 TivaWare_C_Series－2.0.1.11577。

3）将 EK－TM4C123GXL（LaunchPad）开发板的例程导入工作空间，如图 1-7 所示。

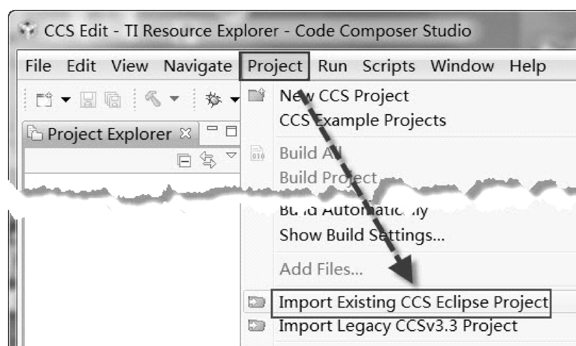


图 1-5　选中 Import Existing CCS Eclipse Project 子菜单

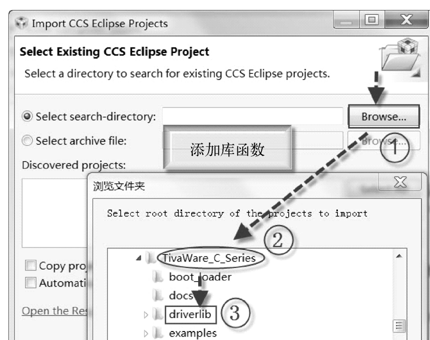


图 1-6　添加库函数的过程

4）编译调试已存在的工程。

① 左键单击某工程来激活该工程，比如激活的 hello 工程如图 1-8 所示。

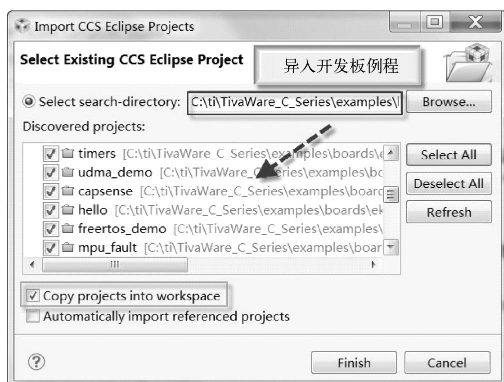


图 1-7　导入 EK－TM4C123GXL 开发板例程

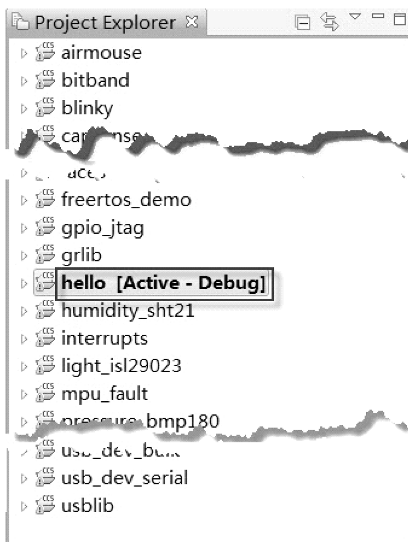


图 1-8　使 hello 工程处于激活状态

② 右键单击 hello 工程，在弹出的下拉菜单中选择 Build Project 编译 hello 工程，其结果如图 1-9 所示。

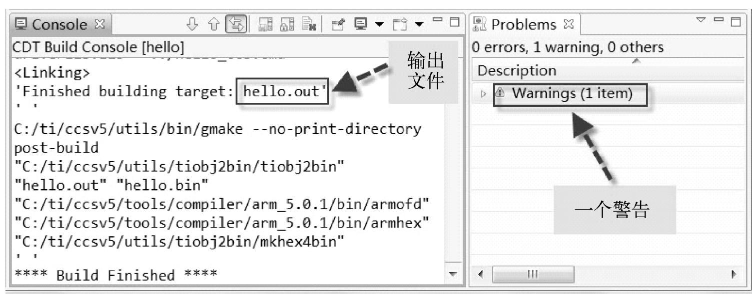


图 1-9　编译结果生成 . out 格式可执行文件

5）在 EK－TM4C123GXL 开发板中运行 . out 文件。

① 创建目标配置文件（. ccxml 文件）。操作步骤：选中 File→New→Target Configuration File 选项，在弹出的 Target Configuration 菜单中给 . ccxml 文件命名为 LaunchPad，如图 1-10 所示。

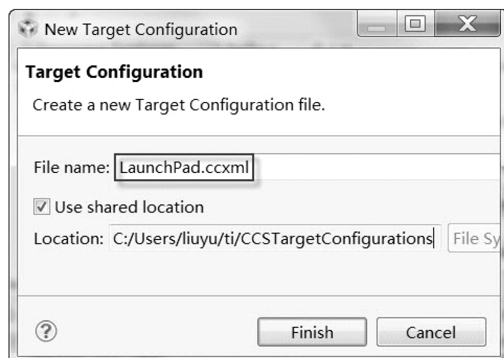


图 1-10　给 . ccxml 文件命名为 LaunchPad

② 在 LaunchPad. ccxml 文件中指定 ARM Cortex 芯片与仿真器，然后按右侧的 Save 按钮保存配置，如图 1-11 所示。

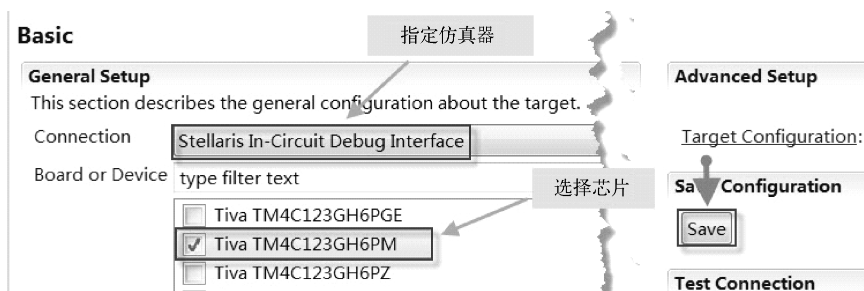


图 1-11　在 LaunchPad. ccxml 文件中指定仿真器与所用芯片

③ 右击 LaunchPad. ccxml 文件，在弹出的下拉菜单中单击 Launch Selected Configuration 选项，发出目标配置信息，如图 1-12 所示。

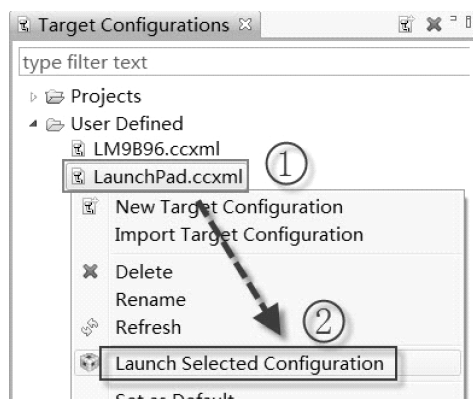


图 1-12　发出目标配置信息

④ 按图 1-13 所示步骤导入 . out 文件并启动程序在 EK - TM4C123GXL 开发板上运行。为了观察在 . out 文件在 EK - TM4C123GXL 开发板中的运行结果，下面将采用 puTTY 串口助手来观察其运行结果。

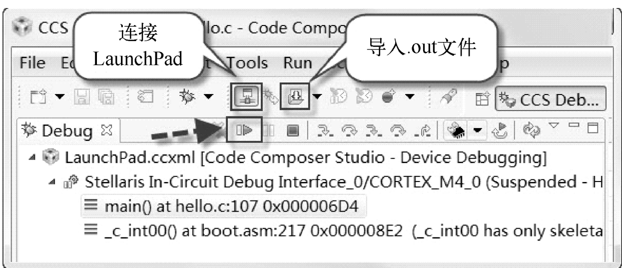
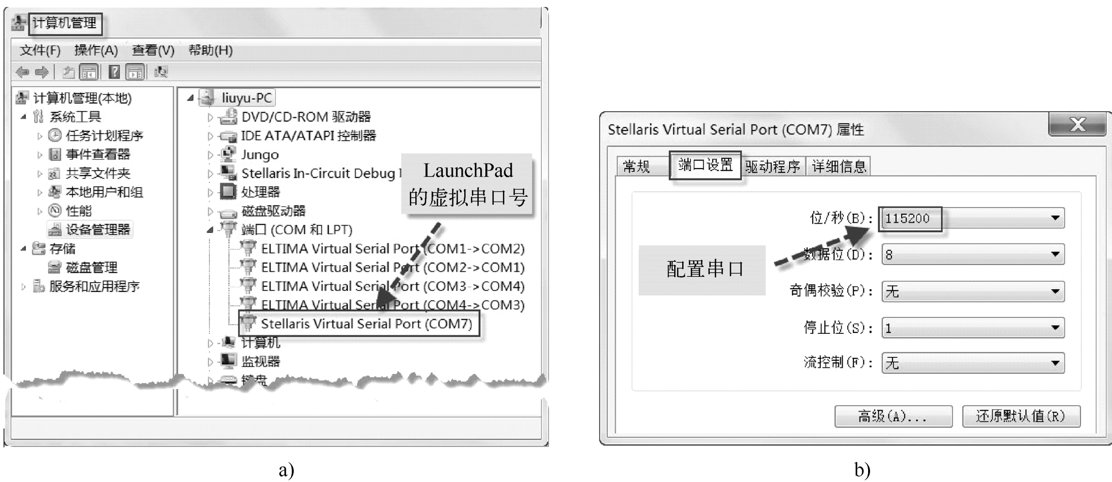


图 1-13　导入 . out 文件的过程及启动程序在 EK - TM4C123GXL 中测试

6）寻找 EK - TM4C123GXL 板子的虚拟串口号与配置串口，如图 1-14a、b 所示。



a)　　　　　　　　　　　　　　　　b)

图 1-14

a）寻找 EK - TM4C123GXL 板子的虚拟串口号　b）配置虚拟串口参数

7）配置 puTTY 串口参数如图 1-15 所示。

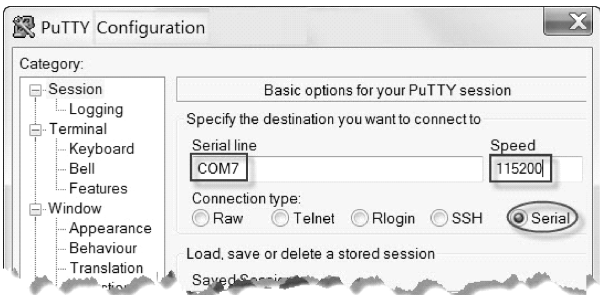
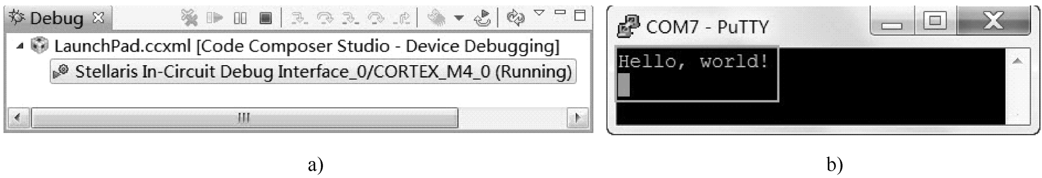


图 1-15　配置 puTTY 串口参数

说明：对于不同的 EK - TM4C123GXL 开发板和电脑，可能分配不同的虚拟串口。

8）重启动 . out 文件在 EK－TM4C123GXL 中运行，在 puTTY 中显示的测试结果，如图 1－16a、b 所示。



a) b)

图 1－16

a）重新启动 . out 文件在 EK－TM4C123GXL 开发板中运行 b）在 puTTY 显示的开发板运行结果

1.2.2 创建一个新工程

1）在主菜单中选择 Project→New CCS Project，在弹出的工作空间创建一个新工程，如图 1－17 所示。

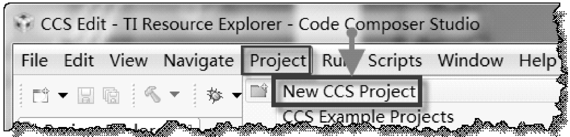


图 1－17 创建一个新工程

2）配置新工程（见图 1－18），然后按 Finish 按钮完成新工程配置，生成的 Myproject 工程如图 1－19 所示。

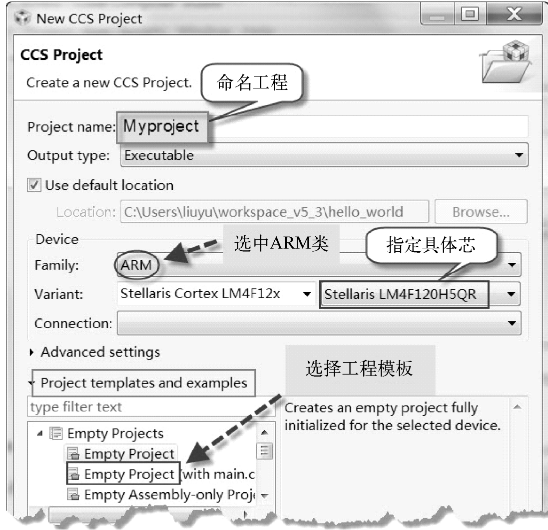


图 1－18 新工程配置

注意：没有特别说明的，接受默认设置。

3）创建或添加 . c 文件到新建的工程中。

① 将 project0 工程中的 project0. c 文件内容全部复制到 main. c 文件中，然后保存该文件。

② 或右键单击 Myproject 工程，在弹出的下拉菜单中单击添加文件（Add Files…）选项（如图 1-20 所示），添加 hello. c 文件，用同样的方式把 startup_ccs. c 文件到工程中，然后删除工程中的 main. c 文件。



图 1-19　新建的工程及自动生成的文件

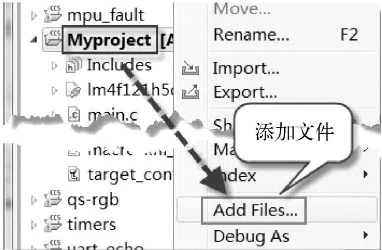


图 1-20　添加文件

注意：对于不涉及中断处理的工程，可以直接把其他工程的 startup_ccs. c 文件复制到新建工程中，或采用创建工程时自动生成 startup_ccs. c 文件；如果所建工程涉及中断处理，则需将函数名添加到中断向量表中。

4）添加包含文件与库文件的搜索路径。添加搜索路径有两种方法：绝对路径与相对路径。

① 绝对路径：把所需的包含文件的绝对路径添加到如图 1-21 所示的栏中。操作步骤为：右键单击 Myproject 工程，在弹出的下拉菜单中选择属性（Properies）选项，然后按图 1-21 编号的步骤进行操作，最后单击 OK 按钮将需要的包含文件路径添加到搜索路径栏中。

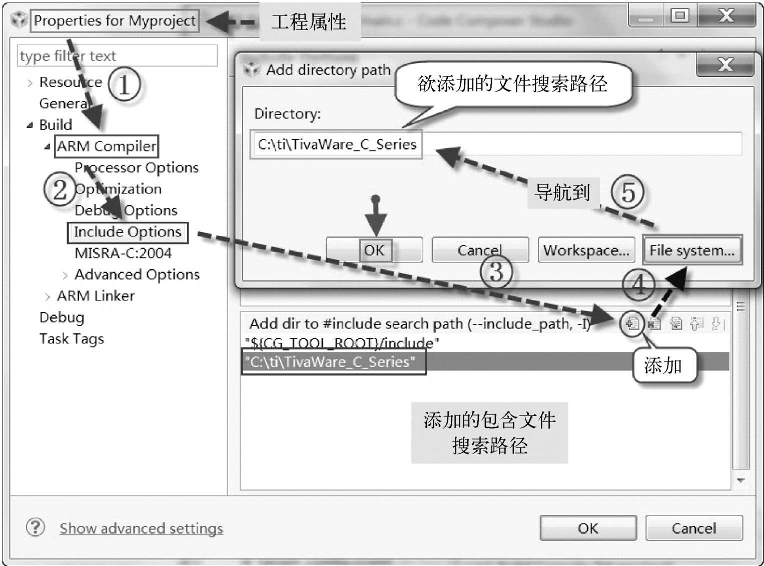


图 1-21　添加包含文件绝对路径的步骤

用同样方法把库文件添加到库文件搜索路径中，如图 1-22 所示。

② 相对路径：

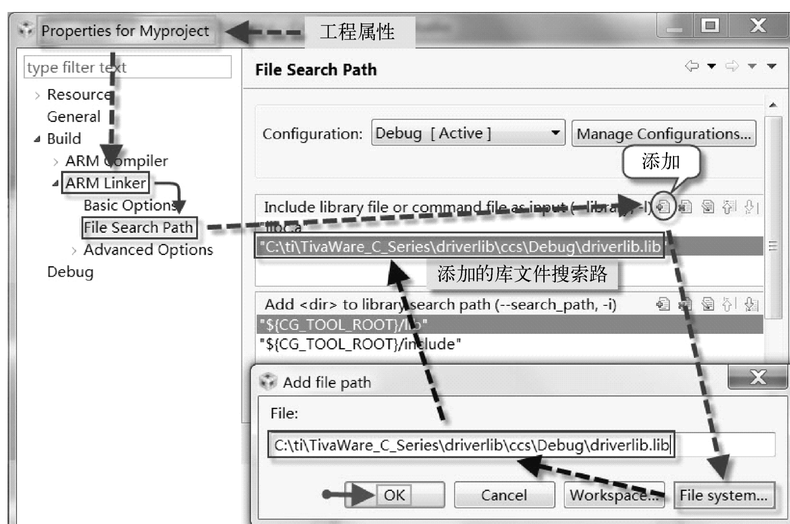


图 1-22　添加库文件绝对路径的步骤

首先在 Build 菜单下选中 Variables 选项，然后单击右边的添加（Add）按钮，创建 ORIGINAL_PROJECT_ROOT 和 SW_ROOT 两个变量，如图 1-23 所示。在搜索路径栏中引用变量产生相对路径，如图 1-24 所示。

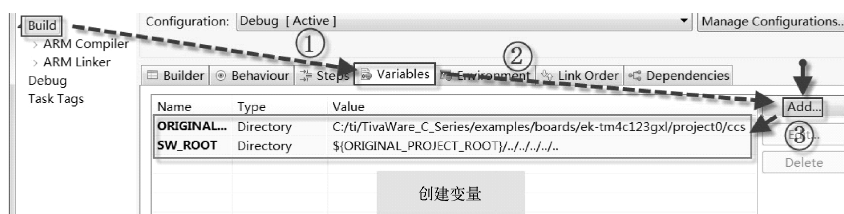


图 1-23　创建变量

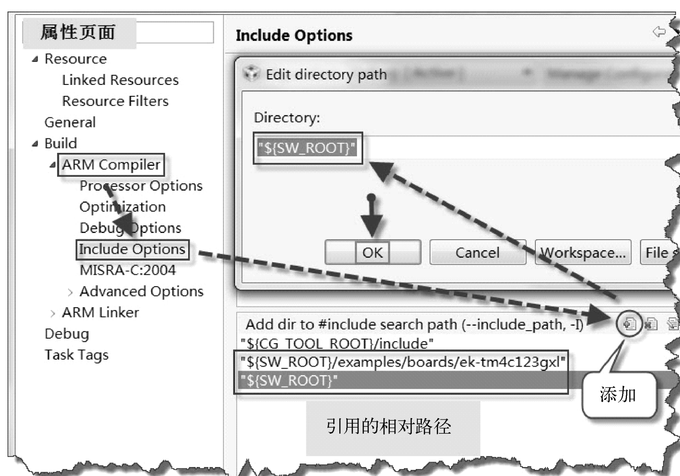


图 1-24　添加包含文件的相对搜索路径

用同样方法添加库文件的相对搜索路径，如图 1-25 所示。

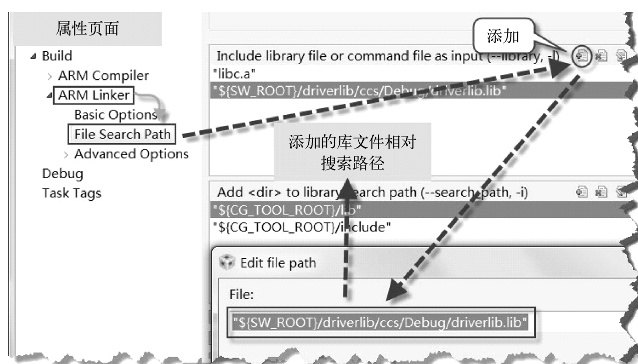
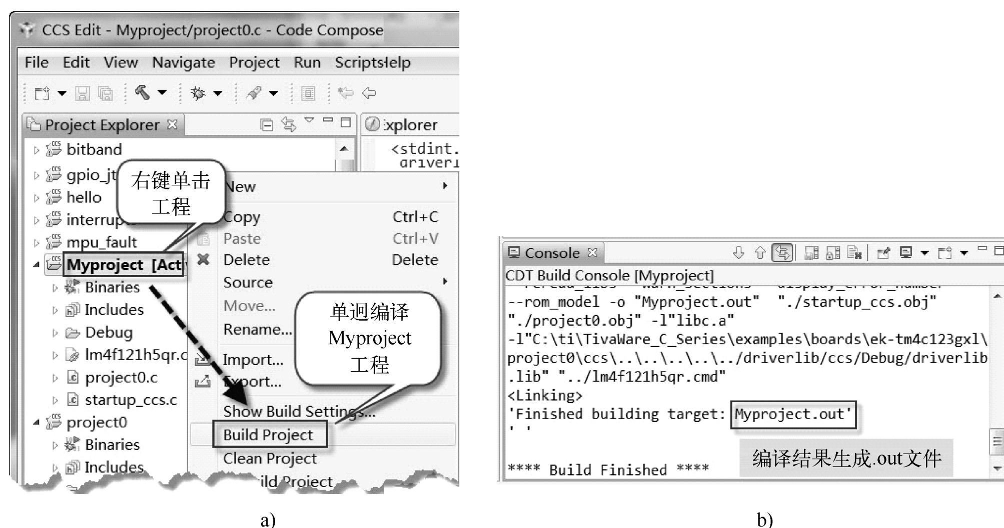


图 1-25　添加库文件的相对搜索路径

5）编译配置成相对搜索路径的 Myproject 工程，其步骤及结果如图 1-26a、b 所示。



a)　　　　　　　　　　　　　　b)

图 1-26

a）编译 Myproject 工程　b）设置成相对搜索路径的 Myproject 工程的编译结果

1.2.3　LM 闪存编程器

本小节将介绍 LM 闪存编程器的基本使用方法及编程示例。

1）LM 闪存编程器是一个独立的 GUI 编程，允许用户通过多个端口编程 Stellaris（Tiva）器件的闪存。

2）确保在 CCS 中没有运行代码的调试透视图，以避免 CCS 和 Flash 编程因 USB 端口控制权发生冲突。单击桌面上如图 1-27 所示的图标，打开闪存编程对话框，如图 1-28 所示。

3）在配置（Configuration）选项的快速设置（Quick Set）栏中选定所需编程的开发板，例如选中 LM4F120 LaunchPad（EK - TM4C123GXL 评估板）。



图 1-27　LM Flash 图标

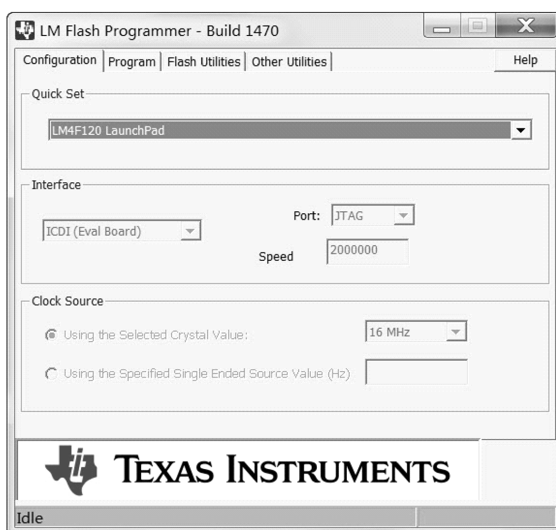


图 1-28　LM Flash 编程器界面

4）这里对 Flash Utilities 和 Other Utilities 选项按默认设置。对这两个选项设置感兴趣的用户请参考右上角的帮助（Help）文件，如图 1-29 所示。

5）单击 Program 选项，对 Options 栏进行如图 1-30 所示的设置。然后按 Select . bin file 栏的浏览（browse）按钮，选择 Myproject 编译生成的 . bin 文件，如图 1-31 所示。

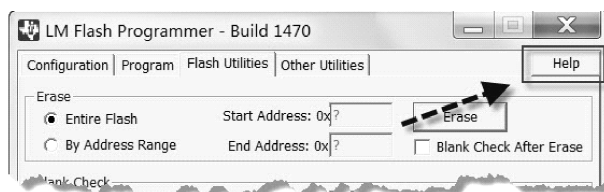


图 1-29　LM Flash 编程的帮助文件

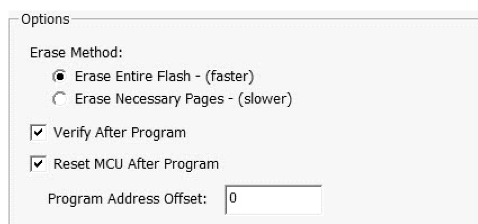


图 1-30　Options 的设置



图 1-31　编译结果中不存在 . bin 格式文件

6）创建闪存编程器的 bin 格式文件。要编译生成 . bin 格式文件必须在 Build 选项下的 Steps 栏中添加以下命令，如图 1-32 所示。

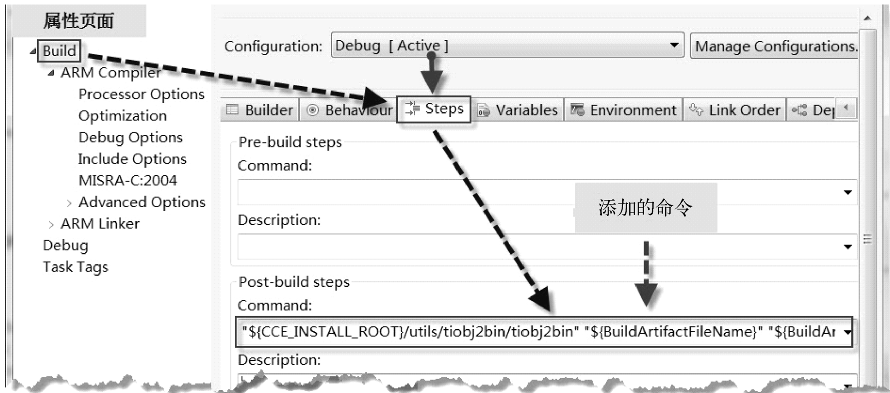


图 1-32　在 Steps 栏中的 Command 选项处添加的命令（见下①、②所述）

① 对于 CCS 5. 2 及更早版本添加以下命令：

" ${CCS_INSTALL_ROOT}/utils/tiobj2bin/tiobj2bin"
" ${BuildArtifactFileName}" " ${BuildArtifactFileBaseName}. bin"
" ${CG_TOOL_ROOT}/bin/ofd470" " ${CG_TOOL_ROOT}/bin/hex470"
" ${CCS_INSTALL_ROOT}/utils/tiobj2bin/mkhex4bin"

② 对于 CCS 5. 3 及更新的版本添加以下命令：

" ${CCE_INSTALL_ROOT}/utils/tiobj2bin/tiobj2bin"
" ${BuildArtifactFileName}"
" ${BuildArtifactFileBaseName}. bin"
" ${CG_TOOL_ROOT}/bin/armofd"
" ${CG_TOOL_ROOT}/bin/armhex"
" ${CCE_INSTALL_ROOT}/utils/tiobj2bin/mkhex4bin"

③ 重新编译 Myproject 工程，其结果如图 1-33 所示。

7）在 LM 闪存编程器的 Program 选项下添加 . bin 格式文件，如图 1-34 所示。



图 1-33　重新编译 Myproject 工程，生成的 . bin 格式文件

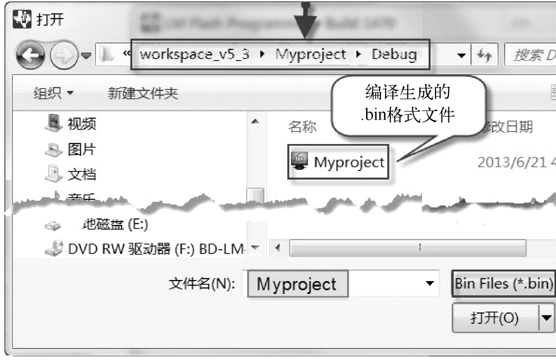


图 1-34　导入 Myproject. bin 文件到 LM 闪存编程器中

8）单击 Program 选项下的 Program 按钮，将 . bin 文件下载到 EK – TM4C123GXL 评估板中，如图 1–35、1–36 所示。

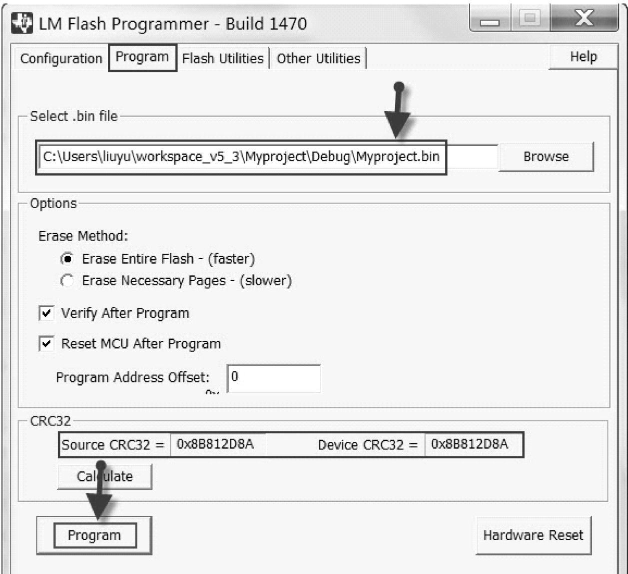


图 1–35　编程 Myproject. bin 格式文件到 EK – TM4C123GXL 评估板中

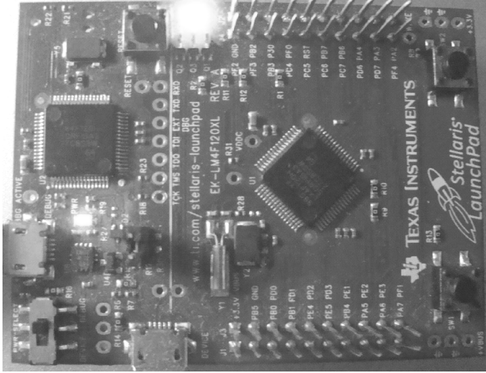


图 1–36　Myproject. bin 文件下载到 EK – TM4C123GXL 评估板中的运行结果



1.3　Keil for ARM 入门基础

本小节将简单介绍 Keil ARM 的使用方法。

作者安装的 MDK – ARM Microcontroller Development Kit 软件是 4. 73 /5. 14 版的，可在其官方网站下载：http://www. keil. com/arm/mdk. asp。

1.3.1　导入一个 hello 工程

1）导入一个 hello 工程。

① 打开 Keil μVision 4 软件，如图 1–37 所示。

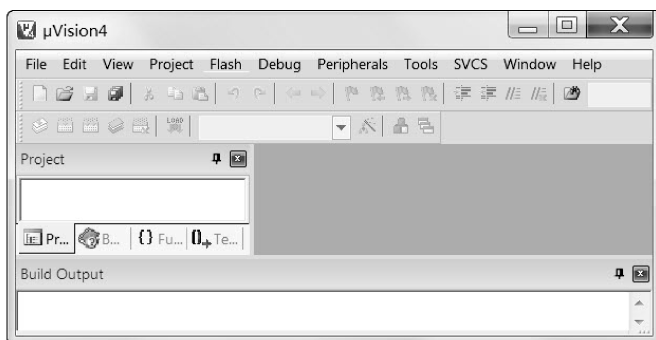


图 1-37 KeilµVision 4 软件界面

② 单击菜单栏中的 Project 菜单，在弹出的下拉菜单中单击“Open Project.” 选项，再在弹出的“工程文件选择对话框”中选中 hello 工程文件，然后按打开按钮导入 hello 工程，如图 1-38a、b 所示。



a) b)

图 1-38
a）导入 hello 工程的过程 b）导入的 hello 工程

2）编译 hello 工程。

① 单击工具栏中的重编译图标，编译 hello 工程，如图 1-39 所示。

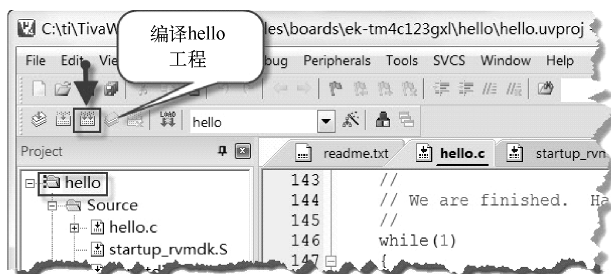


图 1-39 单击“Rebuild all”图标编译 hello 工程

② 在编译时所有源文件将被编译和链接，且在 μVision 4 IDE 窗口的底部将看到这些文件的编译过程。编译完成后将生成以 hello 命名的 . axf 格式文件，如图 1–40 所示。

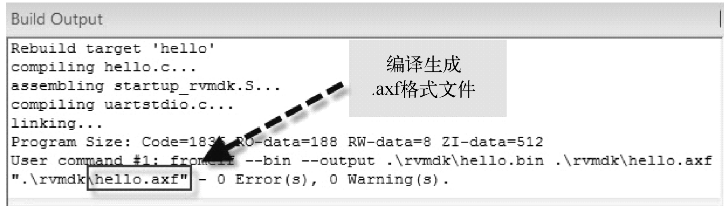


图 1–40　hello 工程的编译结果

3）加载 hello 程序到闪存中。

① 单击工具栏中的下载图标，将对 . axf 格式文件进行编程，其过程会持续几秒，编程结果如图 1–41 所示。

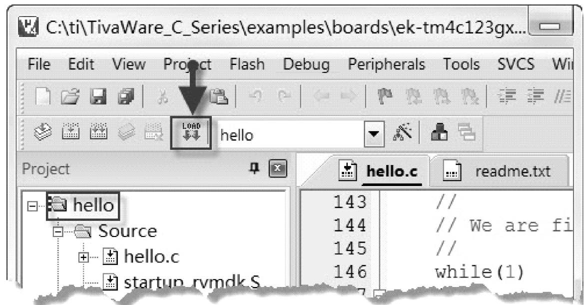


图 1–41　点击图中箭头所指图标对 . axf 文件进行编程

② 此时 hello. axf 文件已下载到 EK – TM4C123GXL 开发板中，如图 1–42 所示。



图 1–42　编程结束并将 . axf 文件下载到 EK – TM4C123GXL 中

4）调试和运行的 hello 程序。

① 选择工具栏中的调试图标，如图 1–43 所示。

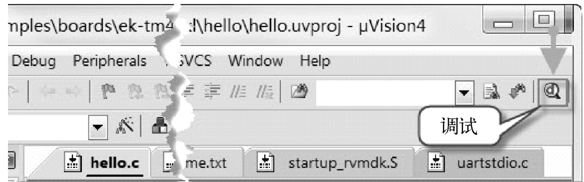


图 1–43　单击调试图标启动代码调试

② 单击调试图标启动 hello 代码调试（此时将打开多种调试窗口），然后单击图中的全速运行按钮，其结果如图 1–44 所示。



图 1-44　多种调试窗口的运行截图

③ 此时 hello 程序的运行结果在 puTTY 中的显示如图 1-45 所示。



图 1-45　hello 程序在 puTTY 中的显示结果

注意：puTTY 串口助手的设置和前面的一致。

1.3.2　创建一个 hello 工程

1）创建 C:\ti\TivaWare_C_Series-2.0.1.11577\examples\boards \my_board 文件夹。

（注：如果读者所建工程找不到搜索路径，可放弃此步骤直接将新建工程放在 C:\ti\TivaWare_C_Series-2.0.1.11577\examples\boards\ek-tm4c123gxl 目录下。）

在此目录下创建 My_ project 工程，其创建过程如图 1-46 ~ 图 1-50 所示。

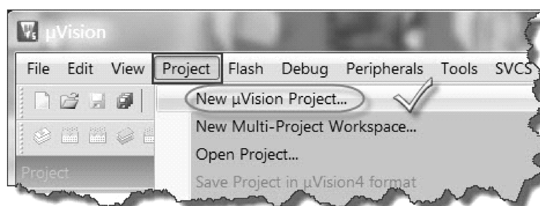


图 1-46　创建新工程

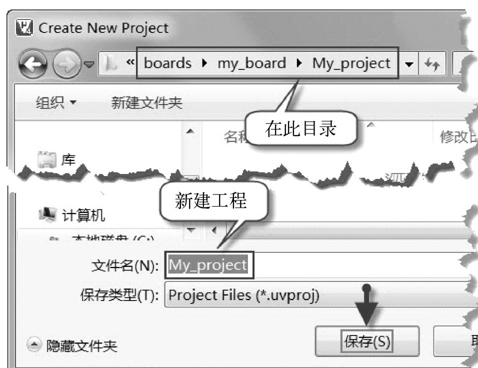


图 1-47　新建 My_project 工程

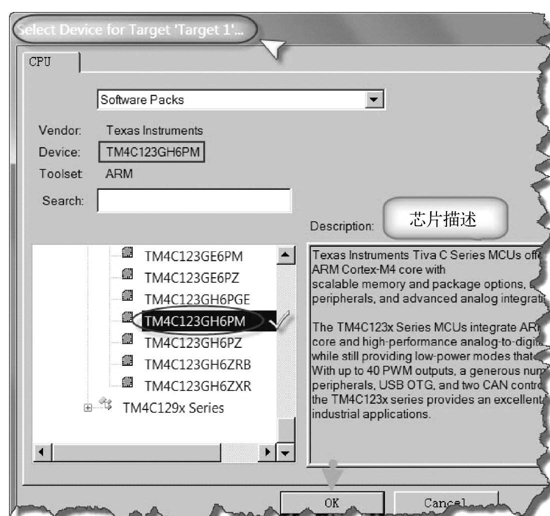


图 1-48　选择工程中使用的芯片型号

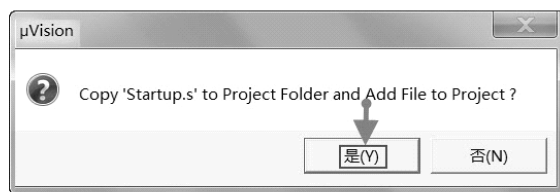


图 1-49　接受生成 Startup. s 文件到工程中

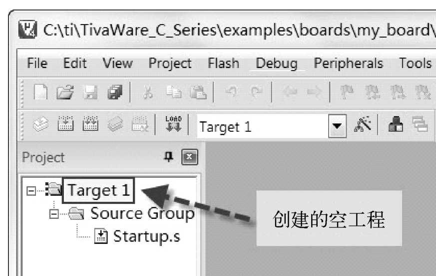


图 1-50　创建的空工程

2）为工程创建与添加源文件，如图 1-51 ~ 图 1-53 所示。

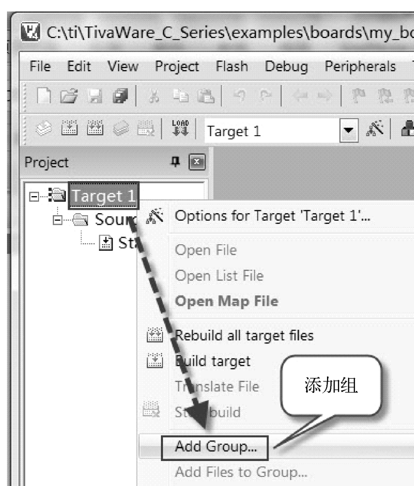


图 1-51　给空工程中添加新组

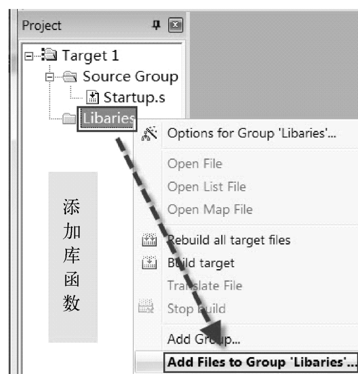


图 1-52　给 Libaries 组中添加库函数

3）配置硬件。首先在工具栏中单击图标，打开 Options for Target ‘Target 1’ 对话框（如图 1-54 所示），再在 Device 选项下选择 TM4C123GH6PM 芯片，然后按图示进行设置，最后单击 OK 按钮确认。

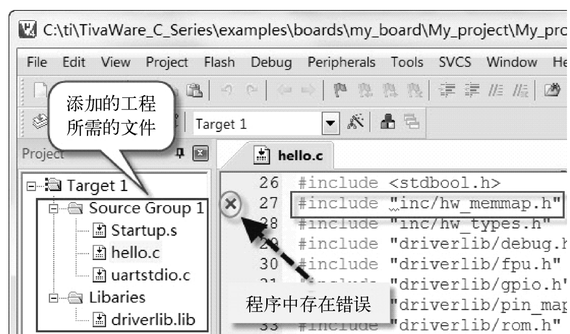


图 1–53　工程添加的源文件与库文件

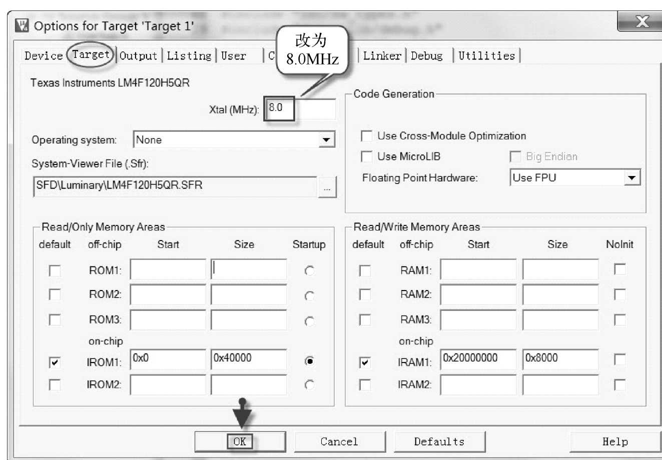


图 1–54　Target 选项配置

4）调试参数配置，如图 1–55 所示。

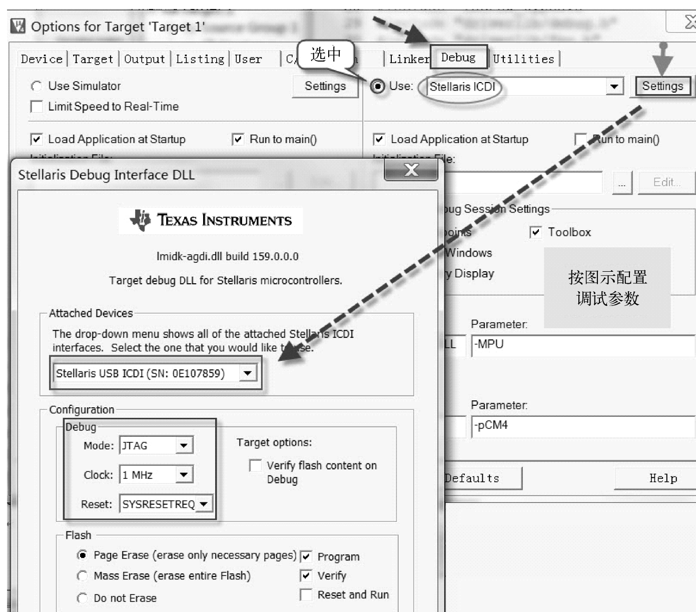


图 1–55　配置调试参数

注意：调试参数按图示配置，未提及的选项按默认设置。

5）按如图 1-56 所示配置闪存编程选项，单击 OK 按钮确定。

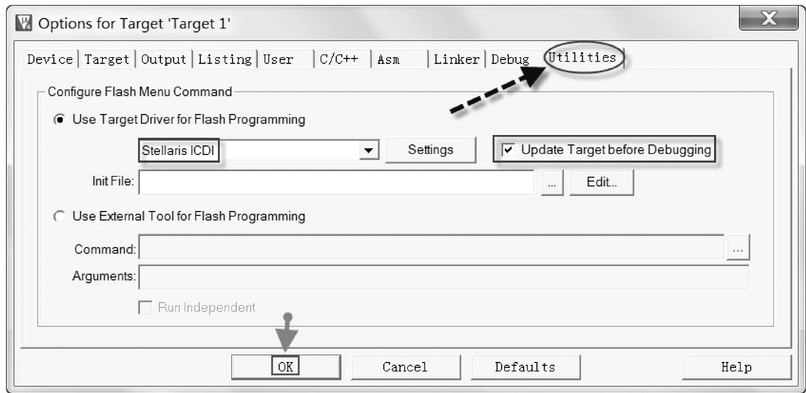


图 1-56　配置闪存编程选项

6）C/C ++ 编译器的配置选项按图 1-57 所示进行，单击 OK 按钮确认设置。

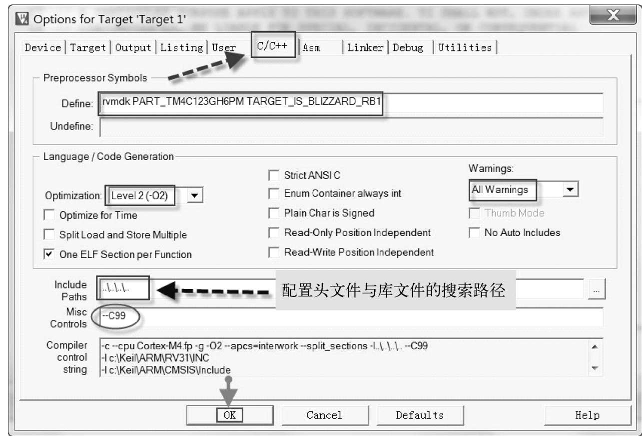


图 1-57　C/C ++ 编译器配置

7）配置“Linker”选项，单击 OK 按钮确认，如图 1-58 所示。

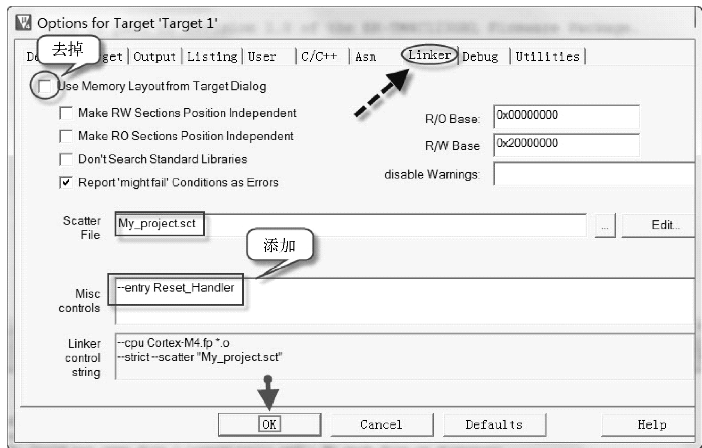


图 1-58　配置“Linker”选项

8）目标 1 选项配置完成后的工程文件，如图 1-59 所示。

9）编译 My_project 工程，然后将 . axf 格式文件编程到 LaunchPad 评估板中，如果此刻已打开了 puTTY 串口助手，将显示“hello world”，如图 1-60 所示。

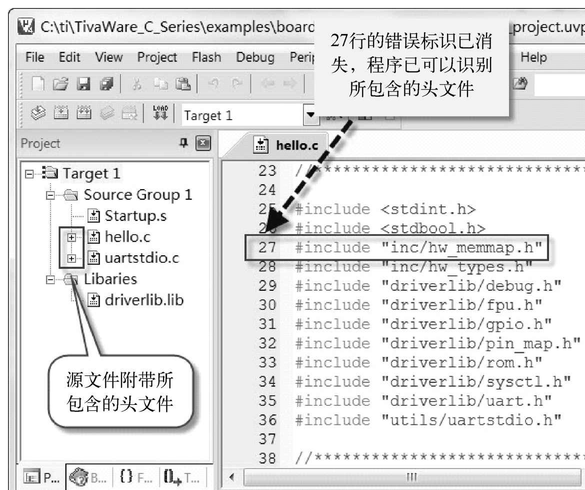


图 1-59　选项配置完成后的工程文件

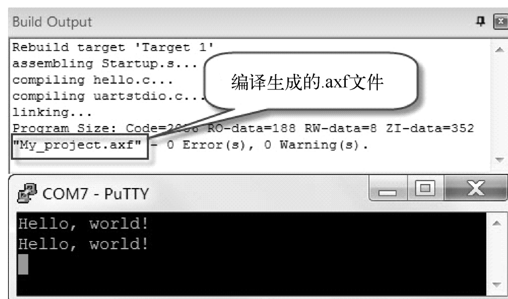


图 1-60　编译与运行结果



1.4　IAR Embedded Workbench for ARM 入门基础

本小节简介 IAR for ARM 的使用方法。

1.4.1　打开一个现有工程

作者安装的 IAR 软件为 7.10 版，可在 http://supp.iar.com/Download/SW/? item = EWARM－EVAL 下载。

1）单击桌面上的图标打开 IAR 软件界面，如图 1-61 所示。



图 1-61　IAR 软件的初始界面

2）单击 File→New→Workspace，打开 IAR 的工作空间，如图 1-62 所示。

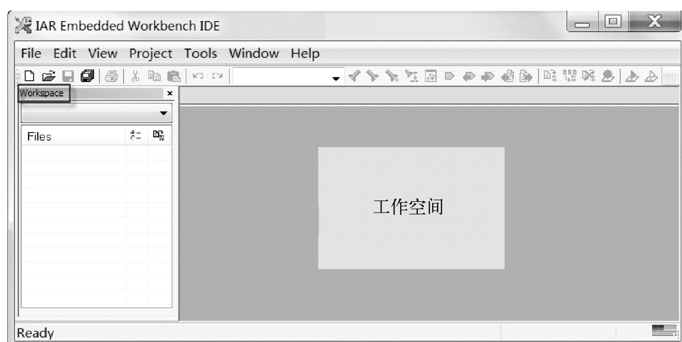


图 1-62　工作空间

3）创建 C:\ti\TivaWare_C_Series-2.0.1.11577\examples\boards \my_board 文件夹。（注：如果读者所建工程找不到搜索路径，可放弃此步骤直接将新建工程放在 C:\ti\TivaWare_C_Series-2.0.1.11577\examples\boards\ek-tm4c123gxl 目录下。）

4）在路径 C:\ti\TivaWare_C_Series-2.0.1.11577\examples\boards\ek-tm4c123gxl 中，导入库文件和例程，如图 1-63 所示。

5）在编译任何工程之前，首先编译 driverlib 库，如图 1-64 所示。

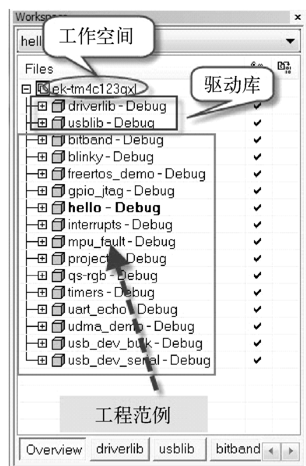


图 1-63　导入的驱动库与例程

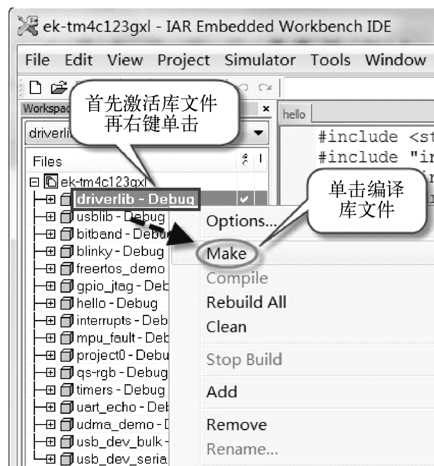


图 1-64　编译 driverlib 驱动库

6）右键单击 hello 工程，然后在弹出的下拉菜单中选择“Set as Active”，激活 hello 工程。再右键单击已激活的 hello 工程，在弹出的下拉菜单中单击“Make”编译驱动库文件，如图 1-65 所示。

7）单击工具栏中的下载与调试图标启动工程调试，此时程序将停止在 main( )处等待执行命令，如图 1-66 所示。

8）用户可以查看和修改存储器、程序变量、处理器寄存器、设置断点、单步执行程序，以及执行其他常用的调试操作。单击工具栏中的全速运行图标，其运行结果如图 1-67 所示。



图 1-65　编译工程

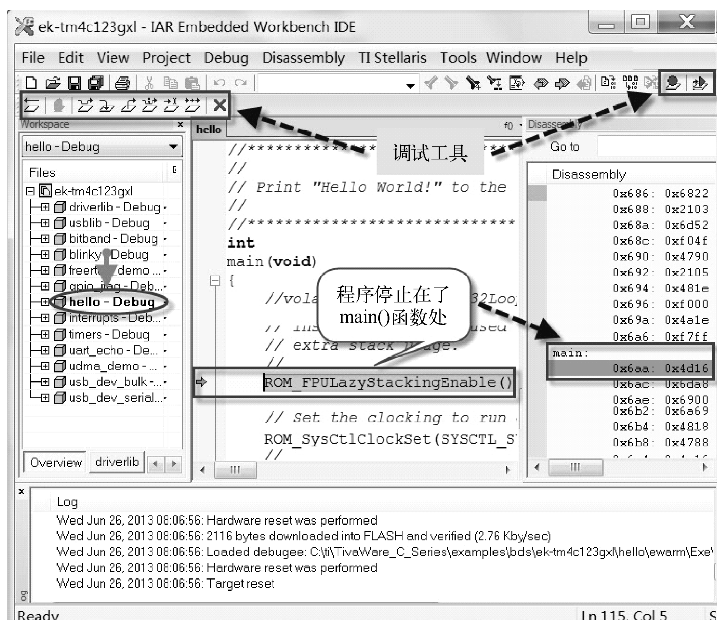


图 1-66　启动工程调试

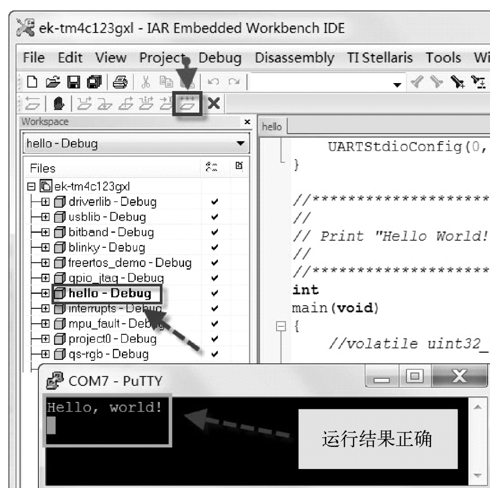


图 1-67　全速运行结果

1.4.2　创建一个新工程

1）单击 File→New→Workspace，新建一个工作空间，如图 1-68 所示。

2）单击 Project→Create New Project，在弹出的菜单中按如图 1-69 进行配置，创建一个新工程。

3）在弹出的对话框中将工程命名为 hello_world，然后单击保存按钮，创建 hello_world 工程，如图 1-70 所示。

最后保存工作空间，将其命名为 my_IAR_project。

4）右键单击 hello_world 工程，在弹出的下拉菜单中选择“Add Group”为工程添加组，如图 1-71 所示。

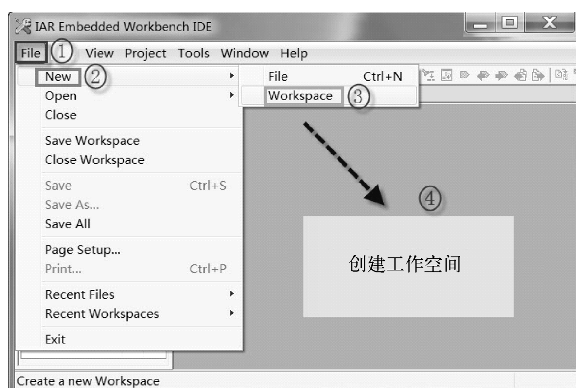


图 1-68　创建工作空间

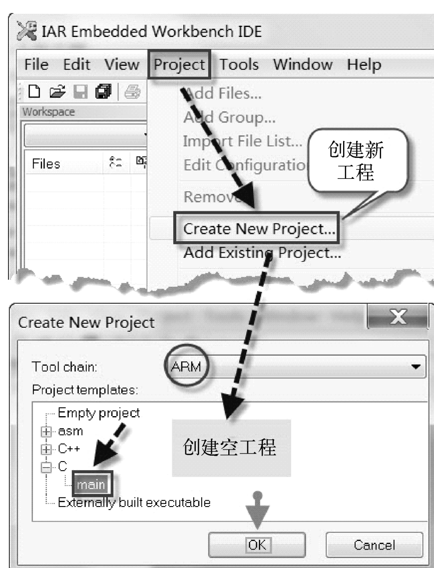


图 1-69　创建新工程



图 1-70　新建的 hello_world 工程

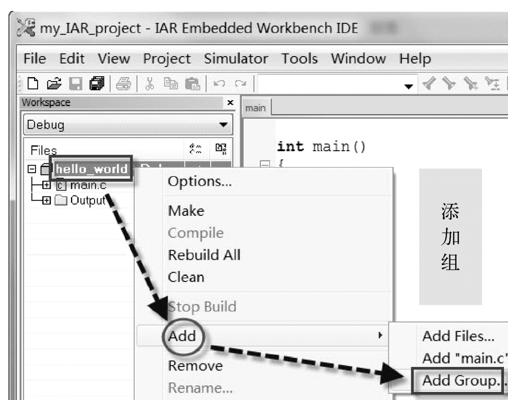


图 1-71　给工程添加组

5）单击“Add Group”，在弹出的对话框中分别敲入 Libraries、Source，然后单击 OK 按钮为工程添加 Libraries、Source 两个组，如图 1-72 所示。

6）用同样的方法为工程添加文件，添加文件后的工程如图 1-73 所示。



图 1-72　为工程添加 Libraries 组

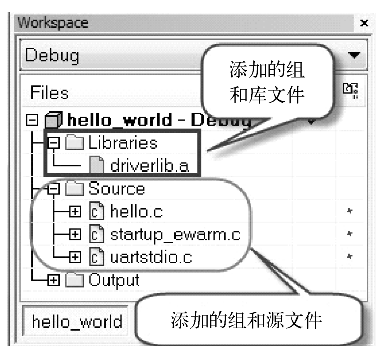


图 1-73　为工程添加组和文件

① 在 C:\ti\TivaWare_C_Series - 2. 0. 1. 11577\driverlib\ewarm\Exe 目录下添加 driverlib. a 库文件。

② 在 C:\ti\TivaWare_C_Series - 2. 0. 1. 11577\utils 目录下添加 uartstdio. c 文件。

③ 在 C:\ti\TivaWare_C_Series - 2. 0. 1. 11577\examples\boards\ek - tm4c123gxl\hello 目录下添加 startup_ewarm. c 和 hello. 。

7）通用选项设置，除了 Target 选项需用户配置外，其他选项可按默认设置即可。

① 选择目标硬件：选中 Device 选项，单击右侧的目标类型图标，选取 LM4F120H5QR 或 TM4C123GH6PM，然后单击 OK 按钮确认，如图 1-74 所示。

② 查看 Output 选项的默认配置，如图 1-75 所示。

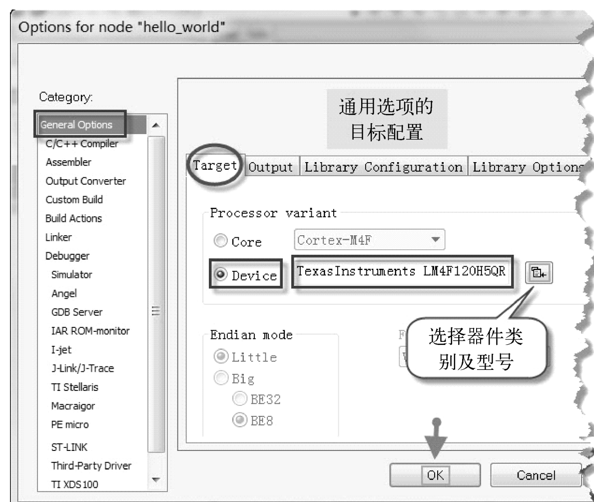


图 1-74　目标硬件配置

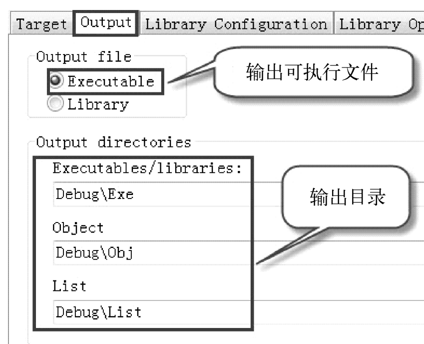


图 1-75　输出选项的默认设置

③ 除了 Preprocessor 选项按如图 1-76 所示的配置外，C/C ++ 选项的其他设置按默认值。

④ Output Converter 选项配置如图 1-77 所示。

⑤ 除了 Setup 选项按如图 1-78 所示的配置外，其他调试选项按默认设置。

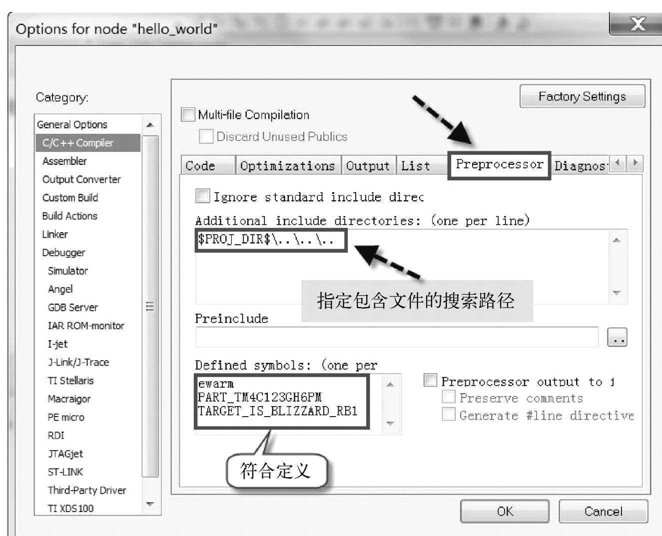


图 1-76　Preprocessor 选项配置

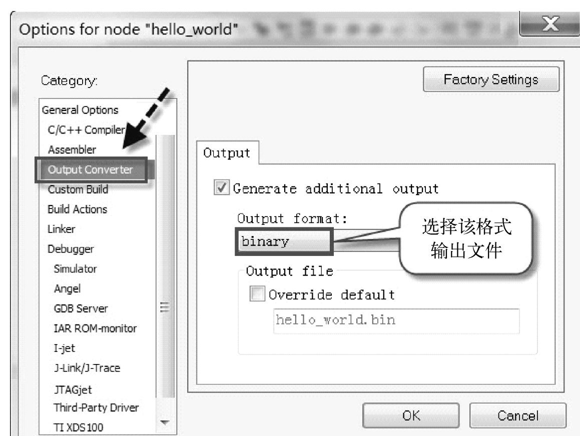


图 1-77　Output Converter 选项配置

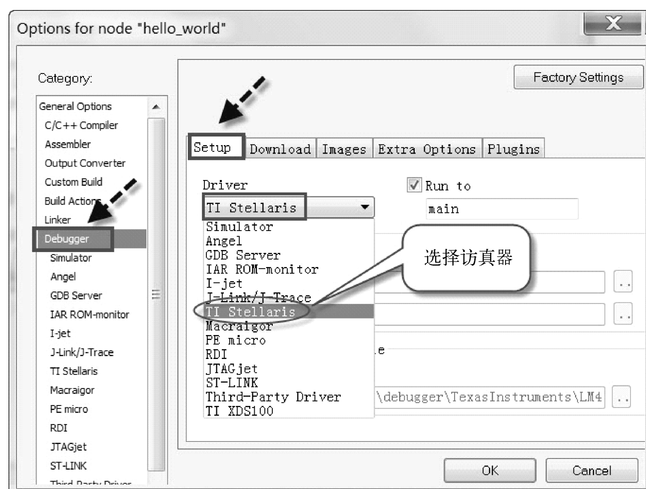


图 1-78　Debugger 选项的 Setup 选项配置

⑥ 在 LaunchPad 中测试程序的运行结果。启动自建工程的调试，如图 1-79 所示。

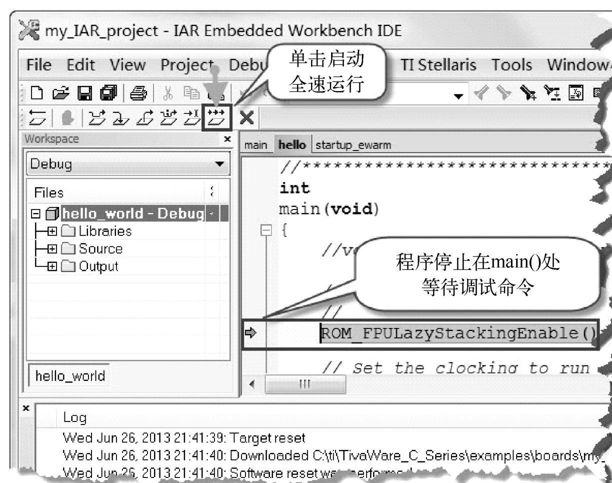


图 1-79　启动自建工程调试

单击箭头所指的全速运行按钮，测试结果如图 1-80 所示。



图 1-80　程序在 LaunchPad 中的运行结果

第 2 章

EK－TM4C123GXL 及 Proteus 简介

如果仅就程序设计与验证方法来看，M4 和 M3 基本上是一致的。为了让那些没有 EK－TM4C123GXL 评估板或缺乏某些外设的读者也能随心所欲的学习 ARM Cortex M4 的编程与测试方法，本章将简述 EK－TM4C123GXL 评估板的结构特点，以及基于 Proteus 8.1 的 M3 程序设计与其测试方法。

本章主要内容：

- EK－TM4C123GXL 评估板简介
- Proteus 8.1 简介
- 虚拟仪器使用方法
- 基于 Proteus 8.1 的 M3 程序设计与测试方法



2.1　EK－TM4C123GXL 简介

Tiva TM4C123G LaunchPad 评估板（EK－TM4C123GXL）（如图 2-1 所示）是一款针对 Tiva TM4C123GH6PM 基于 ARM Cortex－M4 的系列微控制器的低成本评估平台。它突出了 TM4C123GH6PM 微控制器的 USB 2.0 接口、休眠模块和运动控制脉宽调制模块（MC PWM）。它还具有可编程的用户按钮和一个可用于自定义程序的 RGB LED。当连接到许多现有的 boosterpack 附加板的其他外设以及未来的产品，Tiva C 系列 TM4C123G 的 LaunchPad BoosterPack XL 界面可折叠的接头将展示如何轻松地扩展其功能。

2.1.1　TM4C123GXL 的特点

Tiva C 系列的 LaunchPad 包括以下特点：

1）Tiva TM4C123GH6PM 微控制器。

2）运动控制的 PWM。

3）基于 USB micro－A 与 micro－B 连接器的 USB 设备、主机、OTG 连接。

4）RGB 用户 LED。

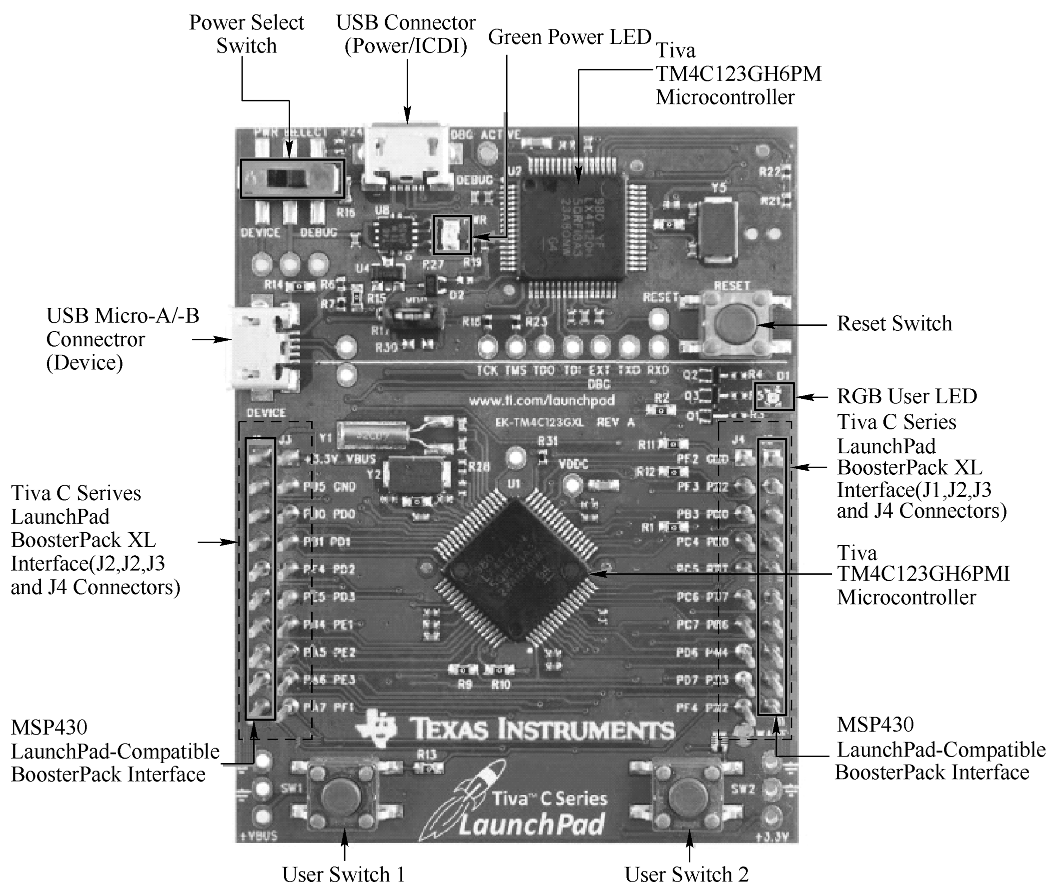


图 2-1　LaunchPad 评估板（EK－TM4C123GXL）

5）两个用户开关（应用程序/唤醒）。

6）引出可用的 I/O 端口，采用公－母连接头间距为 2.54 mm。

7）板载电路内调试接口（ICDI）。

8）通过开关选择电源：

① ICDI。

② USB 设备。

9）复位开关。

10）C 系列 TivaWare 外设驱动库和 USB 库。

2.1.2　评估板模块框图

Tiva C 系列的 LaunchPad 包括 TM4C123GH6PM 微控制器和一个集成的电路内调试接口（ICDI）以及一些有用的外设特性，其模块框图如图 2-2 所示。

更详细的信息请参考 TI 官方文档：“Tiva™ C Series TM4C123G LaunchPad Evaluation Board User's Guide.”

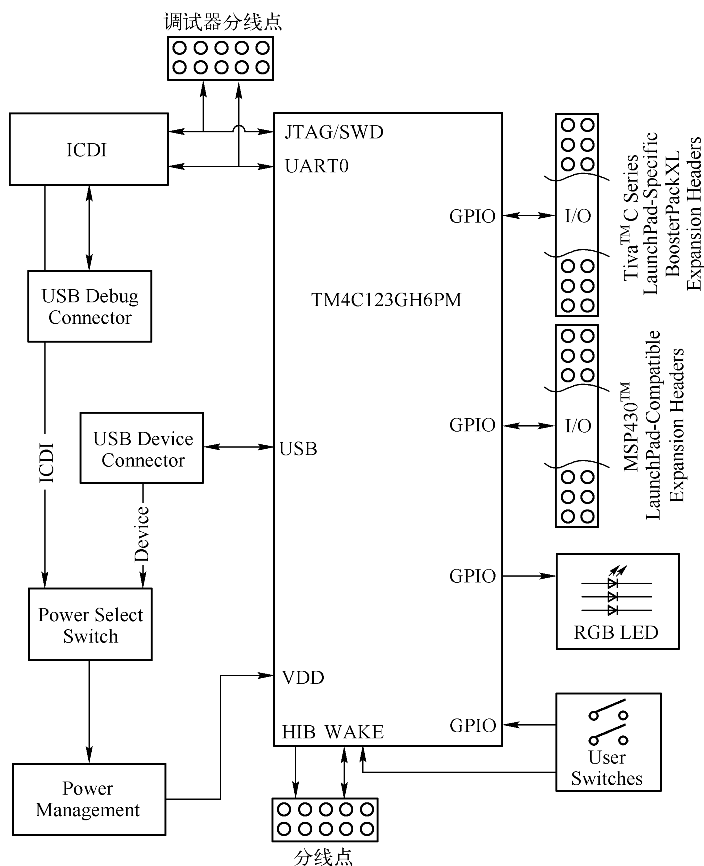


图 2-2 EK-TM4C123GXL 评估板的模块框图



2.2 Proteus 8.1 简介

为了压缩本书的篇幅，这部分仅就 Proteus 8.1 界面、元器件寻找、虚拟仪器使用及一个完全采用 Proteus 8.1 创建、编译及测试的简单 ARM Cortex 工程来介绍其使用与测试方法。有关 Proteus 8.1 的详细信息请参考其官方帮助文档。

注意：只有 Proteus 7.10 或更高版本支持 ARM Cortex M3 芯片。

2.2.1 新增功能

Proterus 8.1 新增功能如下：

① 增加了统一的编译/调试环境，以便调用后台的 Keli、IAR、CCS 等 ARM Cortex 开发工具。

② 新增了 Home Page 方便设计。

③ 新增了 3D 浏览器。

④ 原理图动画功能。

2.2.2 Proteus 8.1 界面简介

单击桌面的图标，可启动 Proteus 8.1 软件，如图 2-3 所示。

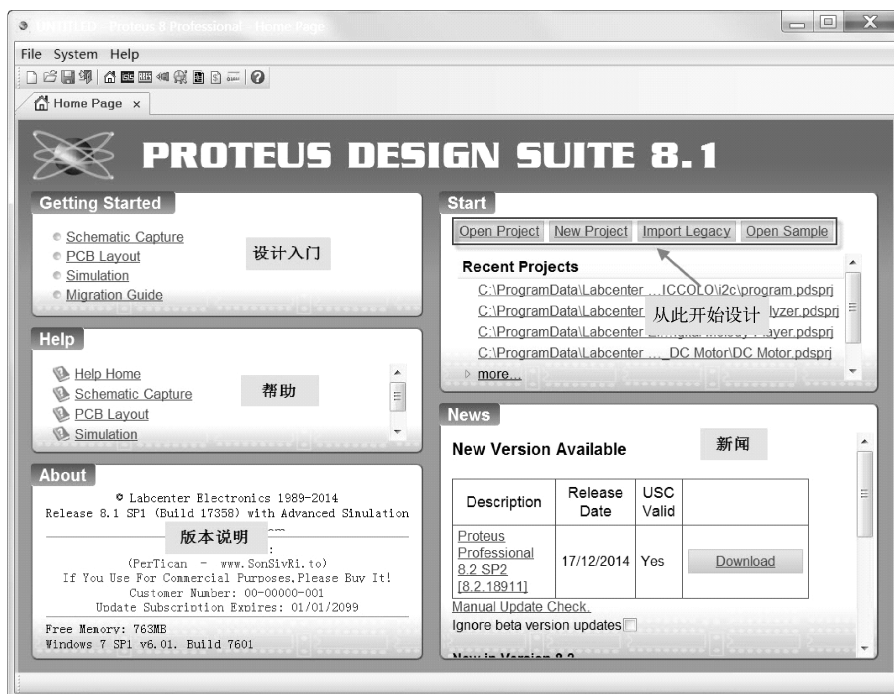


图 2-3　Proteus 8.1 界面

从图 2-3 中可以看到，Proteus 8 比以前版本有较大的改变。

2.2.3　如何寻找 Proteus 中的元器件

1）单击图 2-3 中的图标，打开原理图捕获界面，如图 2-4 所示。

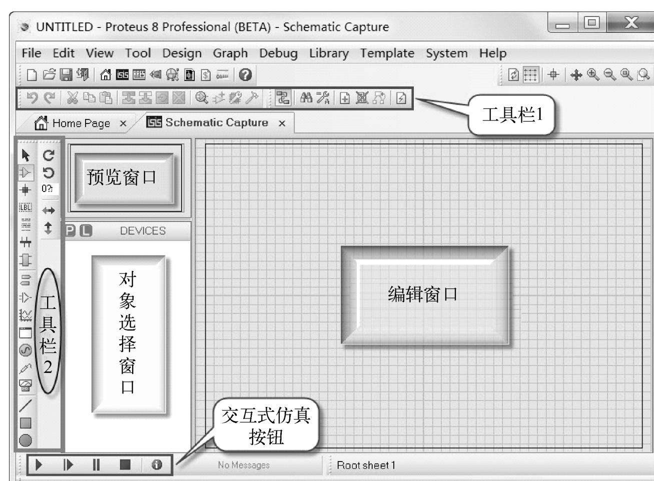


图 2-4　原理图捕获界面

2）在键盘上敲字母 P 或单击预览窗口下面的 P 字母，打开器件选择对话框，在弹出的对话框“keywords”栏中输入 LM3S3，其查询结果如图 2-5 所示。

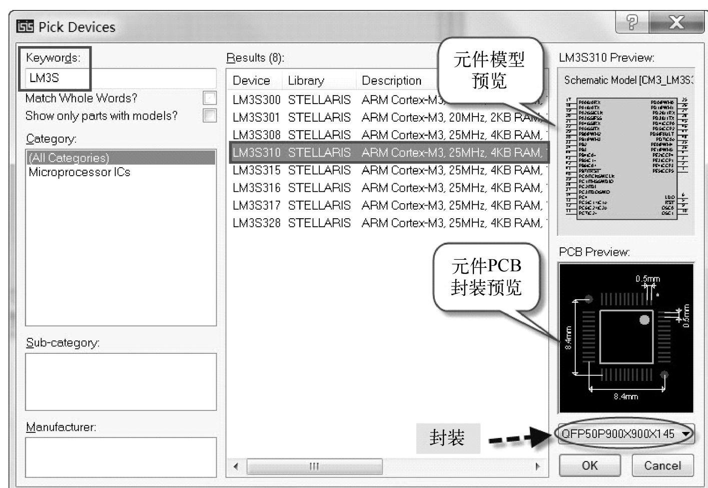


图 2-5　便捷的查找所需元器件方法

注意：还可以根据元器件类别和厂家查找元器件，记住一些常用元器件的英文名称/简称（例如，电阻为 res）是必需的，以便加快元器件的查找。

2.2.4 虚拟仪器的使用

本小节仅介绍与本书有关的虚拟示波器和逻辑分析仪的使用方法。

1. 示波器

VSM 示波器以 ProSPICE 版本为标准，模拟了基本四通道单元及特性，包括：

1）四通道，X—Y 操作。

2）通道增益从 20 V/每格 ~2 mV/每格的 2.5 倍精确设置。

3）时基范围从 200 ms/每格 ~0.5 μs/每格的 2.5 倍精确设置。

4）可以锁定任一通道的自动触发电平。

5）AC 或 DC 耦合输入。

6）A + B 与 C + D 通道模式。

7）每个通道的反转按钮。

8）可通过鼠标缩放。

9）光标测量。

10）单次模式可能的缩放。

11）打印。

12）每个通道可以单独设置颜色。

（1）示波器的使用

单击模式工具栏中的图标，再从对象选择窗口中所列出的所有虚拟仪器中，选中虚拟示波器（OSCILLOSCOPE），这时在预览窗口会显示虚拟示波器的图标，如图 2-6 所示。

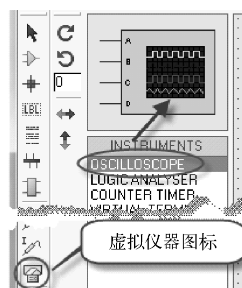


图 2-6　虚拟示波器

在原理图编辑窗口中单击鼠标左键，可放置虚拟示波器到其中。将示波器输入端与被测信号的输出端相连。启动仿真

“开始”按钮，进行交互式仿真，此时，会出现虚拟示波器窗口，并调节扫描频率旋钮到合适的位置。如果显示带直流偏移量的单通道，则选择 AC 模式。调整衰减旋钮和通道位置旋钮获得合适的波形大小和位置。当波形是具有直流电压偏移量的交流信号时，应添加一个隔直电容，调节电平触发旋钮直到显示屏能捕捉到待测的输入波形，如图 2-7 所示。

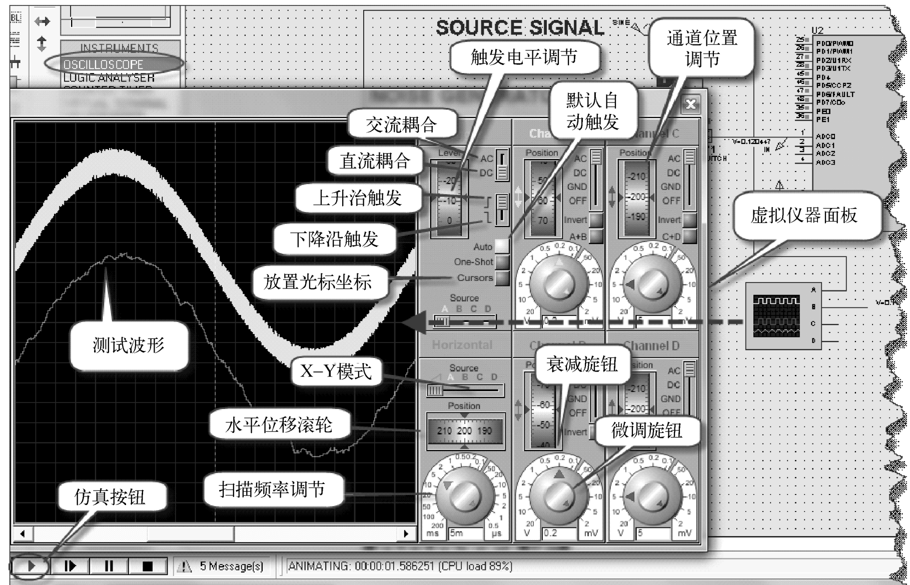


图 2-7　虚拟示波器测试波形示意图

（2）操作模式

示波器有三种操作模式：

① 自动：在这种模式下的自动按钮指示灯点亮，该模式为系统默认。

② 单次：首先关闭自动触发灯。在该模式下单次灯在捕捉过程中被点亮，捕获结束后关闭。

③ X－Y 模式 ：利用滑动条在水平区域选定那个通道（A、B、C、D）作为 X 通道，输出李沙育图。

（3）示波器的触发

① 虚拟示波器具有自动触发功能。这一功能使得输入波形可以与时基同步。

② 通过源（A、B、C、D）滑块在触发区域选择那输入通道用于触发。

③ AC/DC 滑块用于绝对或链接通道的触发偏移量的选择。

④ 电平触发旋钮用于触发偏移量的设置。

⑤ 边缘选择滑块用于波形上升沿与下降沿的选择。

（4）输入耦合

每一通道既可采用直接耦合方式，也可通过仿真电容采用交流耦合方式。其中，交流耦合方式的测量适用于带有较高直流偏压的交流小信号。将输入端临时接地进行校准，这对于测量非常有用。

（5）光标测量

通过鼠标光标可以测量任意点的电压或任意两个点之间的电压差、电流差、时间差。其

步骤为：首先要在触发区域选中光标按钮（cursors），如果欲测量任意点的电压/电流值，只需移动鼠标到待测点单击左键即可；如果测量任意两点间的电压/电流/时间差，可先把鼠标移动到待测的第一个点上，单击左键完成对第一个点的测量，然后再移动鼠标到第二个点上单击左键完成对第二个点的测量，它们之间的差值很容易算出。也可以通过单击鼠标左键选择“Delete Cursor”or“Clear All Cursors”删除这些测试点，如图 2-8 所示。

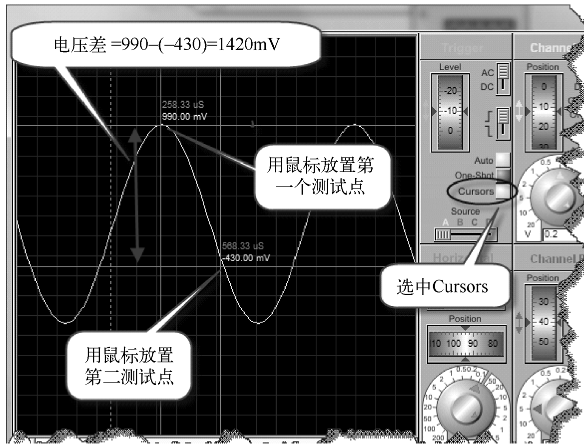


图 2-8　用鼠标光标测量两点间电压差

2. 逻辑分析仪

逻辑分析仪连续地将输入的数字信息记录到一个大的捕获缓冲区。这是一个采样的过程，因此有可调节的分辨率来定义可以被采样的最短脉冲。在触发期间，驱动数据捕捉处理暂停，并监测输入数据。触发前后的数据都可显示。因其具有非常大的捕捉缓冲器（可存放 40000 个采样数据），因此支持放大/缩小显示和全局显示。同时，用户还可移动测量标记，对脉冲宽度进行精确定时测量。

VSM 逻辑分析仪模拟基本的 24 通道单元及特性包括：

1）8 ×1 位通道和 4 ×8 位总线通道。

2）40000 ×52 位捕获缓冲器。

3）捕获分辨率为 0. 5 ns/采样点 ~200 μs/采样点，相应的捕捉时间为 4 s ~10 ns。

4）显示的缩放范围为 1 个采样点/格 ~1000 个采样点/格。

5）输入信号的逻辑电平与/或边沿与总线值进行“与”操作后触发逻辑分析仪。

6）触发位置从 −50% 到 +50%。

7）提供两个坐标用于精确测量时间。

（1）逻辑分析仪的使用

捕获和显示数字数据：

① 在 ISIS 中选中仪表按钮并在对象选择器中选中 LOGIC ANALYSER。放置到原理图编辑窗口中连接到需要测试的地方。

② 单击“开始”按钮启动交互式仿真，逻辑分析仪界面将会出现。

③ 根据需要调节分辨率旋钮到合适位置，分辨率表示了能记录的最小的脉冲宽度。分辨率越高，捕获数据的时间间隔就越短，如图 2-9 所示。

④ 在仪器的左边找到复选框并设置以满足要求的触发条件。例如，如果当连接到通道 0 的信号为高且连接到通道 2 的信号是上升沿时，想要驱动仪器，则需要设置第一位为高，第三位为“low - High”，如图 2-10 所示。

⑤ 根据实际需要，确定是否查看触发发生前后的主要数据，并且调节位置旋钮到需要触发的位置，如图 2-11 所示。

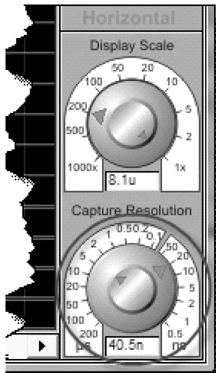


图 2-9　设置分辨率

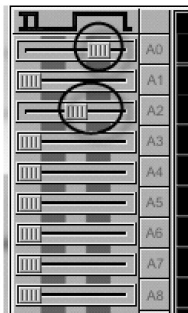


图 2-10　设置触发条件

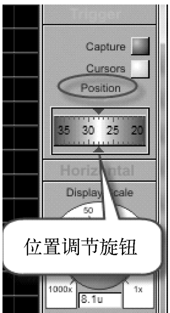


图 2-11　触发位置调节

⑥ 当设置完成后，单击捕获按钮使其变成红色，逻辑分析仪将等待触发事件，并连续捕获输入的数据，同时监控输入触发条件。当触发发生时，捕获灯将变成绿色。数据捕捉将一直进行，直至触发位置之后的捕捉缓冲器满为止。此时，捕捉按钮熄灭，捕获到的数据将出现在显示屏上。

（2）缩放和平移

因为捕捉缓冲器可以捕捉到 10000 个采样点，但是显示屏仅能显示 400 个像素宽，因此需要在捕捉缓冲器进行缩放和平移操作。ZOOM 拨盘决定每格采样点的数量，同时滚动条可以实现左右移动。

2.2.5　基于 Proteus 8.1 的 M3 编程与测试

本小节将介绍采用 Proteus 8.1 软件整合工具，编辑、编译与测试基于 LM3S300 系列的 ARM Cortex 芯片的方法。

1）单击 File→New Project，创建 my_blinky 新工程，如图 2-12 所示。

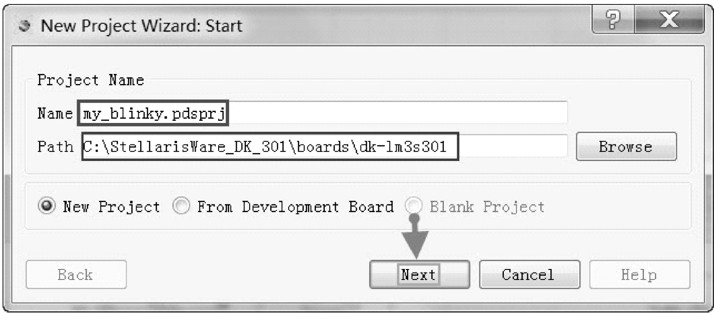


图 2-12　创建 my_blinky 新工程

注意：首先在 C 盘安装 dk-lm3s301 软件包（在随书配套资源中找到），便于 M3 代码的测试。

2）单击图 2-12 中的 Next 按钮，在弹出的对话框中创建工程的原理图，如图 2-13 所示。

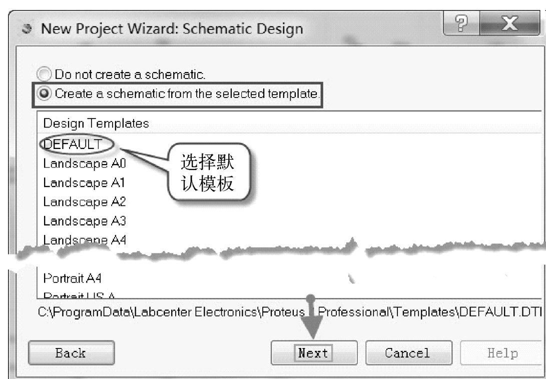


图 2-13　为工程创建一个原理图

3）选中默认模板，然后单击图 2-13 中的 Next 按钮，在弹出的对话框中按图 2-14 所示选择。

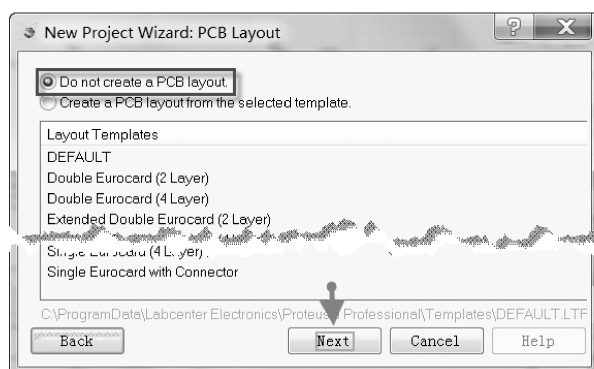


图 2-14　不为工程创建 PCB 版文件

4）单击图 2-14 中的 Next 按钮，在弹出的对话框中为工程选择芯片和后台编译器，如图 2-15 所示。

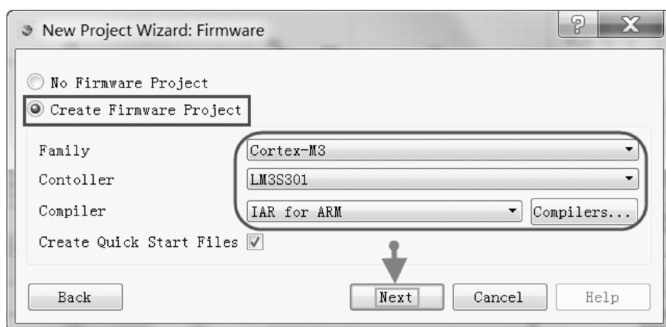


图 2-15　选择芯片及后台编译器

5）单击图 2-14 中的 Next 按钮，弹出创建工程总览对话框，单击 Finish 按钮完成闪烁灯的工程创建，如图 2-16、图 2-17 所示。

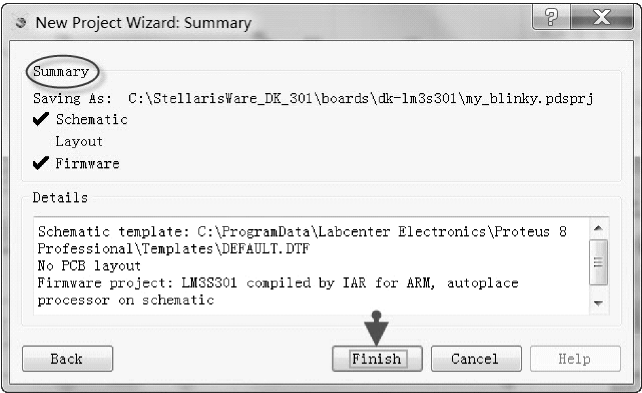


图 2-16　创建工程总览

6）鼠标右键单击工程名，在弹出的下拉菜单中单击 Add Existing 选项为工程添加 startup_ewarm. c 启动代码，如图 2-18 所示，给工程添加闪烁灯和启动代码。

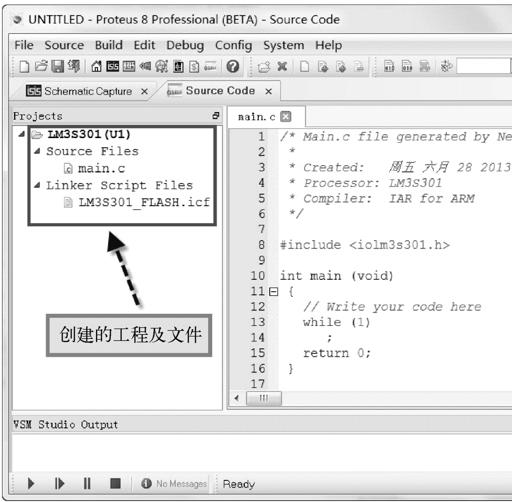


图 2-17　创建的 my_blinky 工程及文件

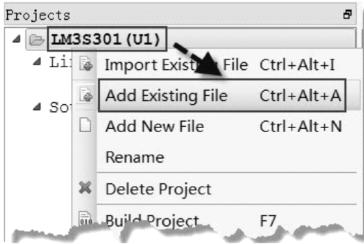


图 2-18　添加启动代码

7）给空 main( ) 函数添加如下的闪烁灯代码。

```
// *****************************************************
#include "inc/lm3s301. h"
int
main(void)
{
    volatile unsigned long ulLoop;      //定义变量
    //
    //使能 PB0 外接的 LED
    //
```

```
        SYSCTL_RCGC2_R = SYSCTL_RCGC2_GPIOB;
        //
        //使能外设后插入几个周期的空读取.
        //
        ulLoop = SYSCTL_RCGC2_R;
        //
        //将 PB0(LED)设置为数字 GPIO 输出
        //
        GPIO_PORTB_DIR_R | = 0x01;
        GPIO_PORTB_DEN_R | = 0x01;
        //
        //死循环
        //
        while(1)
        {
            //
            //点亮 LED
            //
            GPIO_PORTB_DATA_R | = 0x01;
            //
            //延时
            //
            for( ulLoop = 0;ulLoop < 2000;ulLoop ++ )
              {
              }
            //
            //熄灭 LED
            //
            GPIO_PORTB_DATA_R & = ~(0x01);
            //
            //延时
            //
            for( ulLoop = 0;ulLoop < 2000;ulLoop ++ )
              {
              }
        }
    }
```

8）右键单击工程名，在弹出的工程选项设置对话框中，单击编译器（Compiler）选项进行如图 2-19 所示的设置。

9）右键单击工程名，在弹出的下拉菜单中选择 Build Project 编译工程（如图 2-20 所示），其编译结果如图 2-21 所示。

10）搭建闪烁灯程序测试的虚拟硬件电路（如图 2-22 所示），单击图中的 LM3S301 图标，在弹出的对话框中导入 Debug. elf 文件（见图 2-23），然后按 OK 按钮完成 . elf 格式可执行文件的加载。

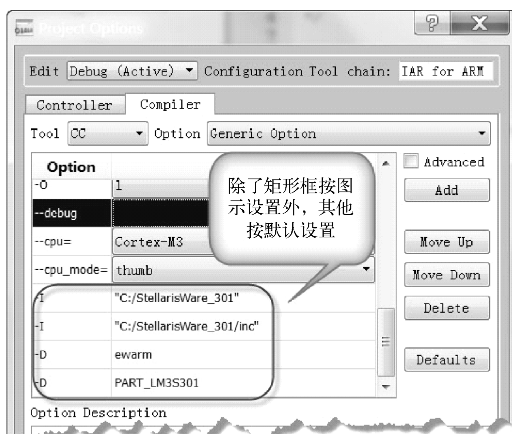


图 2-19　编译器选项配置

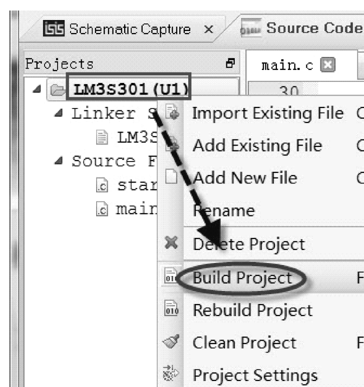


图 2-20　编译工程

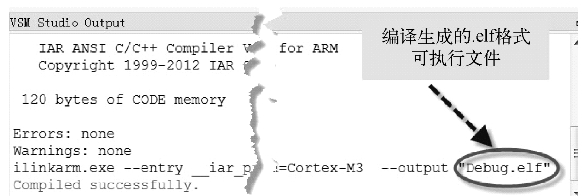


图 2-21　闪烁灯工程的编译结果

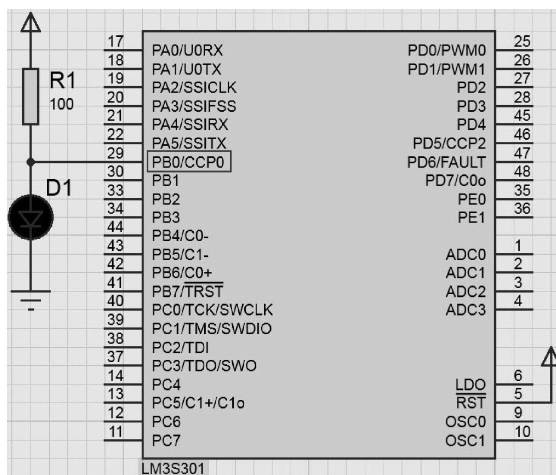


图 2-22　程序虚拟测试电路

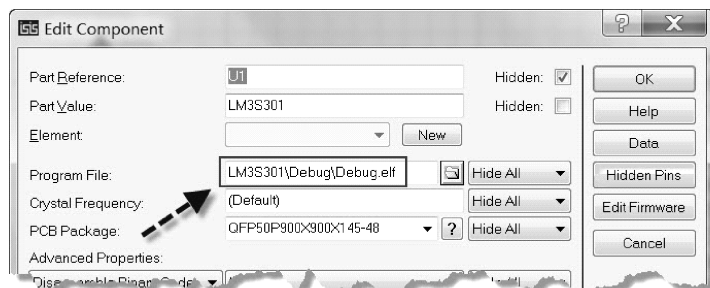


图 2-23　导入 Debug. elf 可执行文件

11）单击原理图中的▶图标，启动虚拟硬件电路的闪烁灯程序测试，其结果如图 2-24 所示。

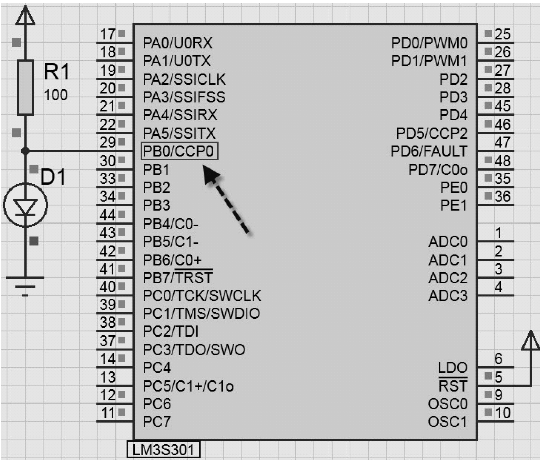


图 2-24　闪烁灯程序的虚拟硬件测试结果

结论：从如图 2-24 中的程序运行结果来看，闪烁灯程序是正确的。采用 Proteus 8.1 可进行编辑、编译与虚拟硬件测试 M3 程序等，不过本书不推荐这种方法。

2.2.6　基于 Proteus 8.1 的 M3 代码测试

1）将 dk - lm3s301 软件包的 blinky 工程导入到 CCS 6 中。

2）将 CCS 的输出文件改成 .elf 格式文件，如图 2-25 所示。

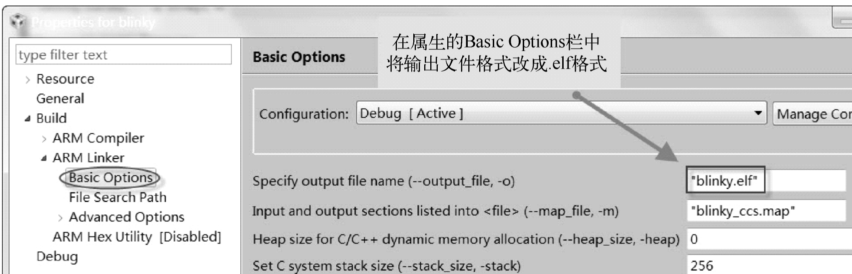


图 2-25　生成 . elf 格式文件

3）编译 blinky 工程，并将生成的 . elf 格式文件导入到 Proteus 测试电路中，如图 2-26 所示。

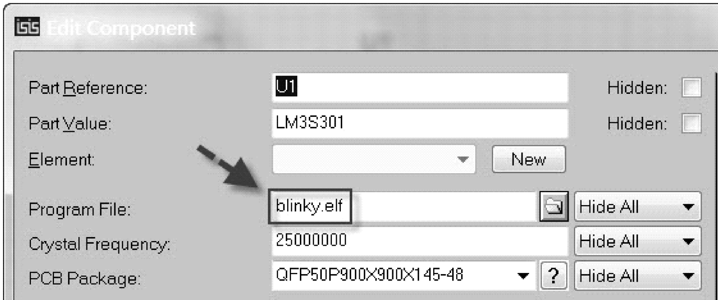


图 2-26　导入 . elf 格式文件

4）制作原理图动画（Schematic Animation），如图 2-27 所示。

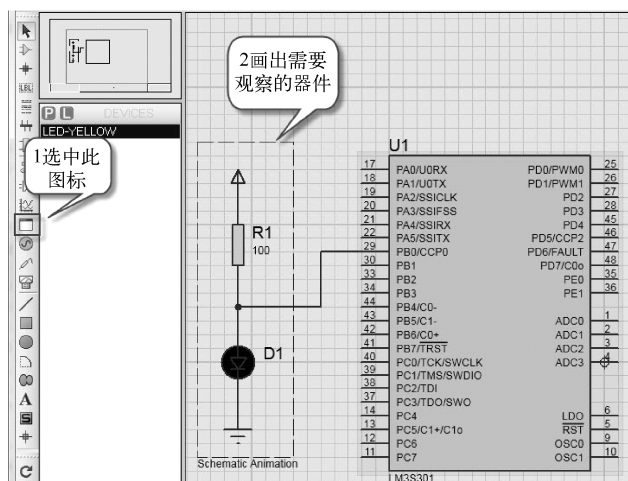


图 2-27　制作原理图动画（Schematic Animation）

5）启动仿真可看到 LED 闪烁，此时可按下暂停键，会出现闪烁灯的源代码，并如图 2-28 所示设置断点。



图 2-28　闪烁灯源代码、原理图动画样式及设置断点

6）在观察窗口中添加变量 GPIO_PORTB_DATA，其步骤为：右键单击 Watch 窗口→在弹出的菜单中选择 Add Items（By Name）选项→选择端口 B→选中 GPIODATA_B→最后单击 Done 完成变量的添加，如图 2-29 所示。

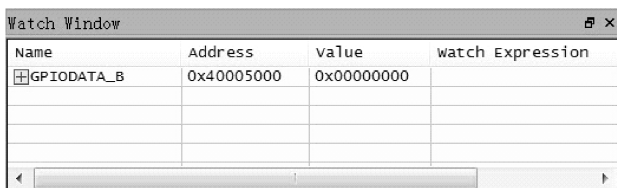


图 2-29　添加观察变量

说明：此时 GPIO_PORTB_DATA 的值为 0，可以看到 LED 灯也没亮，符合程序意图。

7）首先单击调试工具栏上的全速运行按钮，使程序在断点处停止（见图 2-28），然后再单击按钮（Step Over Source Line）或键盘上的 F11 单步运行闪烁灯程序，当执行完断点处（0212）的程序后，LED 灯点亮，同时 GPIO_PORTB_DATA 的值为 1，说明 GPIO_PORTB_DATA_R ｜ = 0x01；语句的确能使 LED 发光，程序正确，如图 2-30 所示。

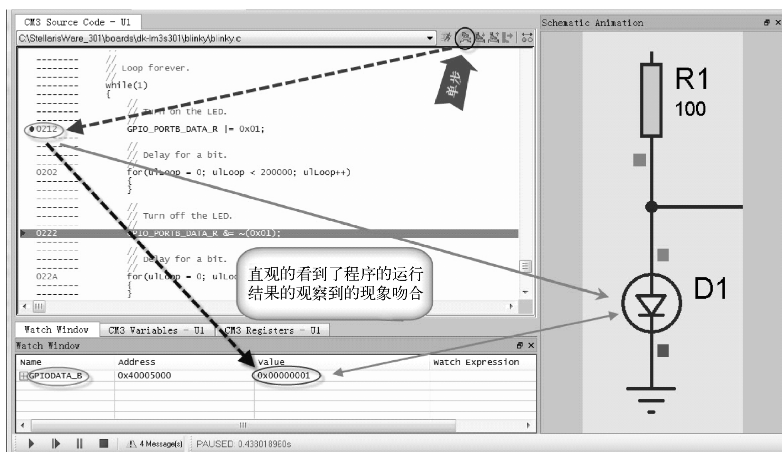


图 2-30　单步运行的结果与观察到的现象吻合（点亮）

8）继续单步，当执行完 0222 语句 GPIO_PORTB_DATA_R & = ~(0x01);后，LED 灯熄灭且 GPIO_PORTB_DATA_R = 0。程序运行结果和观察的现象吻合，该条语句正确。如图 2-31 所示。

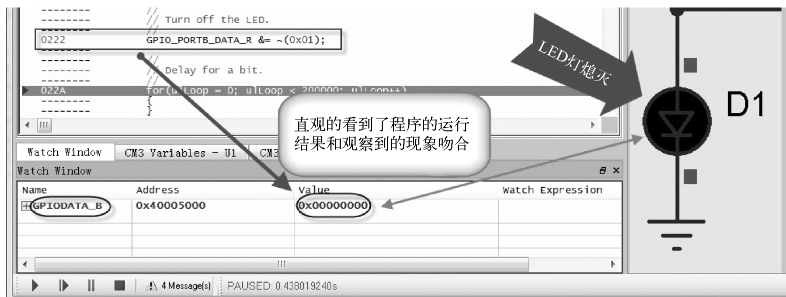


图 2-31　单步运行的结果与观察到的现象吻合（熄灭）

小结：采用 Proteus 的动画调试功能，可轻松查找 M3 代码中存在的错误，或许比在真实开发板卡中进行 M3 程序的调试来得更加容易，值得读者掌握。

第 3 章

通用异步收发器模块（UART）

通用异步接收器/发送器（Universal Asynchronous Receiver/Transmitter，UART）是设备间进行异步通信的关键模块，用于数据总路线与串行接口间串→并、并→串的转换，其优点在于通信双方只需在相同的帧格式和波特率下，无须共享时钟信号仅凭两条数据线就可进行数据通信，并且在完成数据的传输任务之后，可发出中断信号或通过设置标志位来让 CPU 对其进行处理。虽然 UART 的传输速率不高（仅凭两条数据线一位一位的传输），但在一些由微处理器控制的低速设备的通信中却得到了广泛的应用。

TM4C123GH6PM 芯片具有 8 个 UART 单元，其 API 为用户提供了一组配置和控制 UART 模块、发送和接收数据、中断管理功能的函数，其驱动程序包含在 driverlib/uart. c 中，driverlib/uart. h 包含了应用程序所使用的 API 定义。

本章的主要内容：

- UART 模块简介
- UART 固件库函数（见附录 A）
- 例程



3.1 UART 模块

为了控制本书的篇幅，本小节（包括后面的章节）仅给出 TM4C123G 芯片的 UART 模块/及后面各章相关模块的基本介绍，更详细的内容请参考 UART 通信协议/相关模块运行机制及 TI 的技术文档：Tiva™ TM4C123GH6PM Microcontroller DATA SHEET。

3.1.1 UART 的特点

TM4C123GH6PM UART 模块的主要特性如下：

1）可编程的波特率发生器，在常规模式下可高达 5 Mbit/s（16 分频）和在高速模式最高可达 10 Mbit/s（8 分频）的速率。

2）独立的 16×8 发送 FIFO 和接收 FIFO，可降低中断服务程序对 CPU 的占用。

3）FIFO 长度可编程，包括提供传统双缓冲接口的 1 字节深度的操作。

4）FIFO 触发深度分为 1/8、1/4、1/2、3/4 和 7/8 几个等级。

5）标准异步通信位，包括起始位、停止位、奇偶校验位。

6）线中止的产生和检测。

7）完全可编程的串行接口特性：

① 5、6、7、8 个数据位。

② 偶校验、奇校验、stick 或无奇偶校验位的产生/检测。

③ 产生 1 或 2 个停止位。

8）IrDA 串行红外（SIR）编解码器：

① 可编程使用 IrDA 串行红外（SIR）或 UART 输入输出。

② 支持 IrDA SIR 编解码功能，在半双工模式时数据传输速率高达 115.2 Kbit/s。

③ 支持标准的 3/16 和低功耗（1.41 ~ 2.23 μs）的位持续时间。

④ 可编程的内部时钟发生器，允许对基准时钟进行 1 ~ 256 分频来得到低功耗模式的位持续时间。

9）支持 ISO 7816 智能卡的通信模式。

10）在 UART1 模块上的调制解调器流量控制。

11）支持 9 位的 EIA - 485。

12）标准的 FIFO 深度和传输结束中断。

13）高效的微型直接内存访问（μDMA）数据传输：

① 单独的发送与接收通道。

② 当接收 FIFO 中有数据时产生单次请求，当接收 FIFO 达到已编程的触发深度时会产生突发请求。

③ 当发送 FIFO 中有空闲单元时产生单次请求，当发送 FIFO 达到已编程的触发深度时会产生突发请求。

3.1.2 UART 的结构框图

UART 的模块框图如图 3-1 所示。

3.1.3 信号描述

表 3-1 列出了 UART 模块的外部信号并描述了每个信号的功能。UART 信号为表中 GPIO 信号的复用功能，复位时默认为 GPIO 信号（但 U0Rx 与 U0Tx 引脚默认为 UART 功能）。表中的“复用引脚/赋值”一栏列出了各 UART 信号所对应的 GPIO 引脚。可通过对 GPIO 备用功能选择寄存器（GPIOAFSEL）中的 AFSEL 位置位来选择 UART 功能。括号中的数字表示必须写入的 GPIO 端口控制寄存器（GPIOPCTL）中 PMCn 位字段的编码，以便向指定的 GPIO 端口引脚分配 UART 信号。

表 3-1 UART 信号（64LQFP）

引脚名	引脚号	引脚复用/引脚赋值	引脚类型	缓冲区类型①	描 述
U0Rx	17	PA0（1）	I	TTL	UART 模块 0 接收
U0Tx	18	PA1（1）	O	TTL	UART 模块 0 发送
U1CTS	1529	PC5（8） PF1（1）	I	TTL	UART 模块 1 清除发送调制解调器流量控制的输入信号
U1RTS	16 28	PC4（8） PF0（1）	O	TTL	UART 模块 1 的发送请求调制解调器流量控制输出线

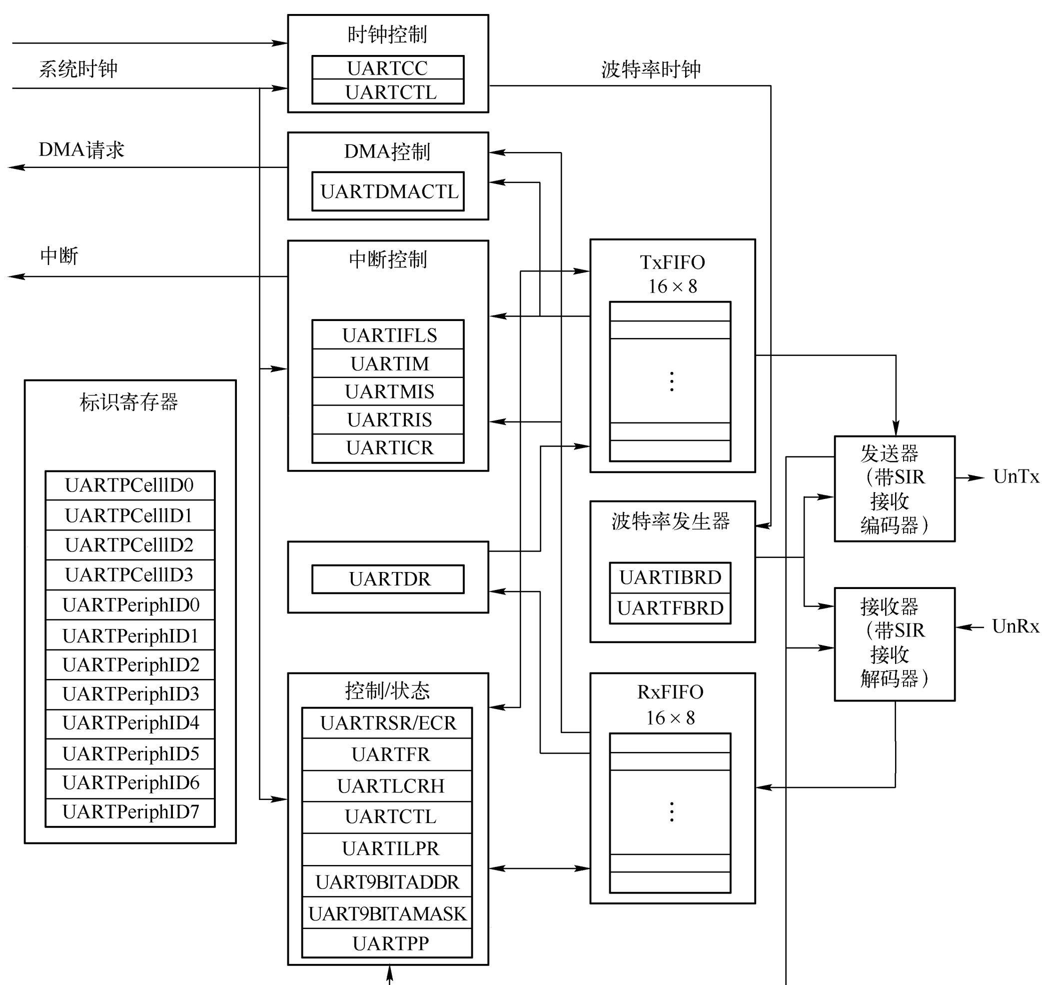


图 3-1　UART 模块框图

（续）

引脚名	引脚号	引脚复用/引脚赋值	引脚类型	缓冲区类型①	描　　述
U1Rx	16 45	PC4（2） PB0（1）	I	TTL	UART 模块 1 接收
U1Tx	15 46	PC5（2） PB1（1）	O	TTL	UART 模块 1 发送
U2Rx	53	PD6（1）	I	TTL	UART 模块 2 接收
U2Tx	10	PD7（1）	O	TTL	UART 模块 2 发送
U3Rx	14	PC6（1）	I	TTL	UART 模块 3 接收
U3Tx	13	PC7（1）	O	TTL	UART 模块 3 发送
U4Rx	16	PC4（1）	I	TTL	UART 模块 4 接收
U4Tx	15	PC5（1）	O	TTL	UART 模块 4 发送
U5Rx	59	PE4（1）	I	TTL	UART 模块 5 接收

（续）

引脚名	引脚号	引脚复用/引脚赋值	引脚类型	缓冲区类型①	描　述
U5Tx	60	PE5（1）	O	TTL	UART 模块 5 发送
U6Rx	43	PD4（1）	I	TTL	UART 模块 6 接收
U6Tx	44	PD5（1）	O	TTL	UART 模块 6 发送
U7Rx	9	PE0（1）	I	TTL	UART 模块 7 接收
U7Tx	8	PE1（1）	O	TTL	UART 模块 7 发送

① TTL 表示该引脚具有与 TTL 兼容的电平。

注意：以后各章该段描述不再给出。

3.1.4 UART 模块功能的简要介绍

下面将简要介绍 UART 模块的功能。

1. 发送/接收逻辑

发送逻辑对从发送 FIFO 读取的数据执行“并 – 串”转换。控制逻辑输出串行位流时，最先输出起始位，然后依据控制寄存器中的配置顺序输出若干数据位、奇偶校验位和停止位，如图 3–2 所示。

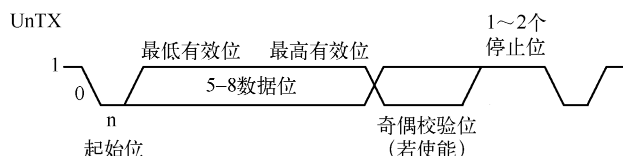


图 3–2　UART 数据帧格式

当检测到有效的起始脉冲后，接收逻辑会对接收到的位流执行“串 – 并”转换。此外还会进行溢出、奇偶校验与线路中止检测以及帧错误检查，并将这些状态随数据一并写入接收 FIFO 中。

2. 波特率产生

波特率除数（baud – rate divisor）是一个 22 位数，它由 16 位整数和 6 位小数组成。波特率发生器由这两个值组成的数字来决定位周期，通过带小数波特率除法器，UART 可以生产所有标准的波特率。波特率除数（BRD）和系统时钟的关系如下：

$$BRD = BRDI + BRDF = SystemClock / (ClkDiv \times BaudRate)$$

其中，BRDI 为 BRD 的整数部分；BRDF 为小数部分；SystemClock 为系统时钟；ClkDiv 为 16（若 UARTCTL 寄存器 中的 HSE 位清零）或 8（若 HSE 位被置位）。

6 位小数部分的计算方法为

$$UARTFBRD[DIVFRAC] = integer(BRDF * 64 + 0.5)$$

UART 会产生一个内部波特率基准时钟，其频率为波特率的 8/16 倍。该基准时钟经过 8/16 分频后即可生产发送时钟，并且它在接收操作期间可用于检测错误。

3. 数据传输

接收或发送的数据被保存在两个 16 字节的 FIFO 中，但接收 FIFO 中的每个字符需包含额外的 4 位状态信息。发送时，数据先被写入到发送 FIFO 中。若 UART 已被使能，则数据帧将按 UARTLCRH 寄存器中所设置的参数启动发送，直到发送 FIFO 中的数据全部发送完成为止。一旦将数据写入到发送 FIFO 中，UART 标志寄存器中的 BUSY 位将生效，且在数据发送期间一直保持有效。BUSY 位仅在发送 FIFO 为空且包括停止位在内的最后一个字符已从移位寄存器中移出时，才会变为无效。这样，即使 UART 不再处于使能状态，也可能被指示为“忙”状态。

在接收器空闲（UnRx 信号持续为 1）且数据输入变为“低电平”（收到起始位）时，接收计数器开始计数，并依照 UARTCTL 寄存器中 HSE 位的配置，在 Baud16 的第 8 个周期或 Baud8 的第 4 个周期对数据进行采样。

若 UnRx 信号在 Baud16 的第 8 个周期（HSE 位清零）或者 Baud8 的第 4 个周期（HSE 置位）仍然为低电平，则起始位有效且可识别，否则可忽略该起始位。在检测到有效起始位后，将根据设定的数据字符长度和 UARTCTL 寄存器中 HSE 位的状态，每 16 个 Baud16 周期或每 8 个 Baud8 周期对后续数据位进行一次采样。若奇偶校验模式被使能，则接着检查奇偶校验位。数据长度和奇偶校验都可在 UARTLCRH 寄存器中定义。

若 UnRx 信号为高电平，则可确认停止位有效，否则可指示发生了帧错误。在接收到一个完整的字时，相关数据与该字相关的错误位将一并被存放到接收 FIFO 中。

4. 串行红外（SIR）

UART 外设包含一个 IrDA 串行红外编/解码器模块。IrDA SIR 模块的功能可在异步 UART 数据流和半双工串行 SIR 接口之间转换。SIR 模块的作用仅为向 UART 提供数字编码输出和解码输入，不会在片上执行任何模拟信号处理。在使能 SIR 模块后，该模块将用 UnTx 与 UnRx 引脚执行 SIR 协议，这些信号应与红外收发器连接来完成 IrDA SIR 的物理层连接。SIR 模块虽可接收和发送数据，但只能以半双工方式进行红外通信，故而不可同时进行接收和发送操作，并且 IrDA SIR 物理层规定在发送和接收之间至少存在 10 ms 的延迟。

SIR 模块有以下两种工作模式：

1）在标准 IrDA 模式下，输出引脚上发送的逻辑 0 电平为高脉冲，其位宽为所选波特率位周期的 3/16，而发送的逻辑 1 电平则是一个静态低电平信号。这些电平控制红外发送器的驱动装置，逢 0 电平便发送光脉冲。在接收端，接收到的光脉冲连接到接收器的光敏晶体管基极，这样将使输出驱动为低电平，并将 UART 输入引脚拉低。

2）在 IrDA 低功耗模式下，变更 UARTCR 寄存器中的相应位后，会将发送红外脉冲的脉宽置为内部产生的 IrLPBaud16 信号周期的 3 倍（在标称频率为 1.8432 MHz 时为 1.63 μs）。

5. 支持 ISO 7816 智能卡的通信模式

UART 为与 ISO 7816 智能卡之间的通信提供了一些基本支持。当 UARTCTL 寄存器中的 SMART 位置位时，UnTx 信号将被作为位时钟信号，而 UnRx 信号则用于连接到智能卡的半双工通信线路。而 GPIO 信号可用于向智能卡发送复位信号，其余智能卡信号应由系统设计提供，在该模式下的最大时钟速率为系统时钟的 1/16。

在使用 ISO 7816 模式时，UARTLCRH 寄存器必须配置为发送 8 位数据字（WLEN 字段 6：5 配置为 0x3），带偶校验（即 PEN 与 EPS 位置位）。在该模式下，UART 将自动使用 2 个停止位，而忽略 UARTLCRH 寄存器中的 STP2 位。

若在发送期间检测到奇偶校验错误，UnRx 将在第二个停止位期间被拉至低电平。在这种情况下，UART 将终止发送、刷新发送 FIFO 中的所有数据，同时产生一个奇偶校验错误中断来让软件检测到该问题，并重新发送受影响的数据。注意，UART 不支持该种情况的自动重发功能。

6. 支持调制解调器握手信号

下面将介绍在 UART 作为数据终端设备（DTE）或数据通信设备（DCE）连接时，如何配置 UART1 和使用调制解调器的流控制信号。一般情况下，调制解调器为 DCE，而连接到调制解调器的计算设备为 DTE。

（1）发信号

根据 UART 是作为 DCE 还是作为 DTE 的不同，UART1 所提供的状态信号也会有所不同。

1）当作为 DTE 时，调制解调器的流控信号定义如下：

① $\overline{\text{U1CTS}}$允许发送信号。

② $\overline{\text{U1RTS}}$请求发送信号。

2）当作为 DCE 时，调制解调器的流控信号定义如下：

① $\overline{\text{U1CTS}}$请求发送信号。

② $\overline{\text{U1RTS}}$允许发送信号。

（2）流控制

流控制既可通过硬件也可通过软件来实现。下面将分别描述这两种方法。

1）硬件流控制（RTS/CTS）。两个设备之间的硬件流控制是由$\overline{\text{U1RTS}}$输出端连接到接收设备的“清除发送（Clear－To－Send）”输入端实现的，以及将“请求发送”的输出端连接到接收设备上的 $\overline{\text{U1CTS}}$输入端上。由$\overline{\text{U1CTS}}$输入端控制发送器，而且只有在$\overline{\text{U1CTS}}$输入端有效时发送器才能发送数据。$\overline{\text{U1RTS}}$输出信号可指示接收 FIFO 的状态，且$\overline{\text{U1CTS}}$将一直保持有效，直到达到预设的电平，此时接收 FIFO 中已无多余空间可用。UARTCTL 寄存器中的 CTSEN 位和 RTSEN 位指定流控模式，见表 3-2。

表 3-2　流控制模式

CTSEN	RTSEN	描　　述
1	1	使能 RTS 和 CTS 流量控制
1	0	只使能 CTS 流量控制
0	1	只使能 RTS 流控制
0	0	禁止 RTS 和 CTS 流量控制

注意：当 RTSEN 为 1 时，软件无法通过 UARTCTL 寄存器的请求发送（RTS）位来修改$\overline{\text{U1RTS}}$的输出值，并且应忽略 RTS 位的状态。

2）软件流控制（调制解调器的状态中断）。通过使用中断来指示 UART 的状态，从而完成两台设备间的软件流控制。使用 UARTIM 寄存器中的第 3 位，以便为$\overline{\text{U1CTS}}$信号发出中断。可使用 UARTRIS 寄存器和 UARTMIS 寄存器查看原始中断和屏蔽中断的状态，并可通过 UARTICR 寄存器来清除这些中断。

7. 9 位 UART 模式

在 9 位模式下，UART 可用于多点通信，并通过 UART9BITADDR 寄存器中的 9BITEN 位来使能。主机可通过其地址（或一组地址）以及一个地址字节限定符（地址字）与某个特殊的从机进行通信。如果所有地址限定符的从机检查置位，可通过比较接收到的字节与预编程的地址，如果地址匹配，那么将接收或发送更多的数据；如果地址不匹配，将丢弃该地址字节和后续的数据字节。可用 UART9BITADDR 寄存器来预定义地址来匹配所接收的字节，以及使用 UART9BITAMASK 寄存器中的地址屏蔽，让匹配扩展到一组地址，默认时，UART9BITAMASK 为 0xFF，意味着只有指定的地址可匹配（注意，在 9 位模式下，接收器操作无奇偶校验模式）。

在未找到匹配时，清零的第 9 位和其余数据字节将被丢弃；如果找到匹配，将会向 NVIC 发出一个中断以便进一步的动作，清零的第 9 位和后续数据字节将被保存到 FIFO 中。若此时使能了 μDMA 和/或 FIFO 操作，软件可屏蔽该中断，而无须处理器干预。9 位模式的所有传送操作都为数据字节且第 9 位被清零。软件可将奇偶校验设置为粘着奇偶校验，对某个字节使能奇校验，以改写置位的第 9 位。欲使发送时间与奇偶校验设置匹配，可将地址字节作为单次传输而不是突发传输。由于发送 FIFO 并不含地址/数据位，因此软件应使能相应的地址位。

8. FIFO 操作

UART 具有 2 个 16×8 的 FIFO：其中一个用于发送，而另一个用于接收。这两个 FIFO 都可通过 UART 数据寄存器（UARTDR）进行访问。UARTDR 寄存器的读操作将返回一个 12 位的值，包括 8 个数据位和 4 个错误标志，而写操作则将 8 位数据保存到发送 FIFO 中。

复位后，两个 FIFO 均处于禁止状态，并作为 1 字节深的保持寄存器。可通过对 UARTLCRH 寄存器中的 FEN 位置位来使能这两个 FIFO。

通过 UART 标志寄存器（UARTFR）和 UART 接收状态寄存器（UARTRSR）来监控 FIFO 的状态，对于空、满和溢出条件则由硬件进行监控。UARTFR 寄存器包含空和满的标志（TXFE、TXFF、RXFE 和 RXFF 位），而 UARTRSR 寄存器则通过 OE 位来指示溢出状态。如果 FIFO 被禁止，空和满的标志由 1 字节深的保持寄存器的状态设置。

FIFO 产生中断的触发点是通过 UART 中断 FIFO 深度选择寄存器（UARTIFLS）来控制。可将两个 FIFO 分别配置为在不同的深度触发中断。可供选择的配置包括 1/8、1/4、1/2、3/4 和 7/8，例如，给接收 FIFO 选择了 1/4，则 UART 会在接收 4 个数据字节之后产生接收中断。未复位时，两个 FIFO 都被配置为在 1/2 的深度触发中断。

9. 中断

在出现以下情况时，UART 将产生中断：

① 溢出错误。

② 中止错误。

③ 奇偶校验错误。

④ 帧错误。

⑤ 接收超时。

⑥ 发送（当满足 UARTIFLS 寄存器中的 TXIFLSEL 位所定义的条件时，或对 UARTCTL 寄存器的 EOT 位置位且所有发送数据的最后 1 位已从串行移位寄存器移出时）。

⑦ 接收（当满足 UARTIFLS 寄存器中的 RXIFLSEL 位所定义的条件时）。

在所有中断事件发送到中断控制器之前，先行对其执行“逻辑或”操作，因此 UART 一次只能向控制器发送一个中断请求。通过读取 UART 屏蔽中断状态寄存器（UARTMIS），软件可在一个中断服务例程中处理多个中断事件。

对 UART 中断屏蔽寄存器（UARTIM）中的相应 IM 位置位，可定义能触发控制器级别的中断事件。如果不使用中断，原始中断状态在 UART 原始中断状态寄存器（UARTRIS）中总是可见的。向 UART 中断清除寄存器（UARTICR）中的相应位写 1 即可清除该中断。

当接收 FIFO 不为空时，接收超时中断有效，在 32 位周期内（在 HSE 位清零时）或在 64 位周期内（当 HSE 位置位时）不再接收数据。接收超时中断既可自动清除（当读出 FIFO 或保持寄存器中的所有数据，使 FIFO 变为空状态时），也可将 UARTICR 寄存器的相应位置位来手动清除。

1）当发生下列事件时，接收中断的状态将被改变：

① 如果 FIFO 使能且接收 FIFO 达到设定的触发点，RXRIS 位被置位。当从接收 FIFO 中读取的数据低于触发点时，将自动清除接收中断，或通过向 RXIC 位写 1 来手动清除中断。

② 如果 FIFO 被禁止（有一个位置的深度）且接收到的数据已填充该位置，RXRIS 位将被置位。通过对接收 FIFO 执行一次读取操作来清零接收中断，或通过向 RXIC 位写 1 来清除中断。

2）当发生下列事件时，发送中断的状态将被改变：

① 如果 FIFO 被使能且发送 FIFO 达到设定触发点，TXRIS 位将被置位。发送中断的产生依据触发点的高低，因此在 FIFO 中写入的数据必须超过该设定的触发点，否则不会再产生发送中断。当向发送 FIFO 中写入的数据大于触发点时来清除发送中断，或通过向 TXIC 位写 1 来清除中断。

② 如果 FIFO 被禁止（有一个位置的深度）且无数据存在于发送器的单个位置，TXRIS 位将被置位。可通过向发送 FIFO 中执行一次写入操作来清除发送中断，或通过向 TXIC 位写 1 来清除中断。

10. 回送操作

可对 UARTCTL 寄存器中的 LBE 位置位，来使 UART 进入内部回送模式，以便 UART 进行诊断和调试。在回送模式下，UnTx 输出端发送的数据将被 UnRx 输入端接收。

注意：需先行对 LBE 位置位，再使能 UART。

11. DMA 操作

UART 向 μDMA 控制器接口提供独立的发送和接收通道。UART 的 DMA 操作通过 UART DMA 控制寄存器（UARTDMACTL）来使能。当 DMA 操作使能后，在相应的 FIFO 传输数据时，UART 将向接收通道或发送通道发出 DMA 请求。对于接收通道，如果接收 FIFO 中含有数据，就将发出单次传输请求。如果接收 FIFO 中的数据量达到或超过 UARTIFLS 寄存器中配置的 FIFO 触发深度，则会发出突发传输请求。对于发送通道，只要发送 FIFO 中

至少有一个空位，就将发出单次传输请求。如果发送 FIFO 中所含的字符少于 FIFO 触发深度，则会发出突发请求。并且 μDMA 控制器将根据 DMA 通道的配置自动处理单次和突发 DMA 传输请求。

如果使能接收通道的 DMA 操作，需对 DMA 控制寄存器（UARTDMACTL）中的 RXDMAE 位进行置位。如果使能发送通道的 DMA 操作，需对 UARTDMACTL 寄存器中的 TXDMAE 位进行置位。还可将 UART 配置为在发生接收错误时，使接收通道停止使用 DMA。如果对 UARTDMACR 寄存器的 DMAERR 位进行了置位且发生接收错误，则会自动禁止 DMA 接收请求。如遇到此种情况可通过清除相应的 UART 错误中断来清除。

若使能了 μDMA，则 μDMA 控制器将会在传输完成后自动触发中断。在 UART 操作中使用了中断且使能了 DMA，那么在 UART 中断处理函数中必须包含对 μDMA 完成中断的处理。



3.2 UART 固件库函数

下面将简要介绍 UART 固件库的结构和常用的几个 API 函数及其基本操作，完整 UART 固件库函数的参数定义、功能描述等请参看书后附录 A 第 3 章附录。

3.2.1 UART 固件库结构

UARTAPI 一般包含四个功能组：

① 配置和控制。

② DMA 操作。

③ 发送和接收。

④ 中断处理。

1）由以下函数完成 UART 的配置和控制：

➢ UARTConfigGetExpClk( )。

➢ UARTConfigSetExpClk( )。

➢ UARTDisable( )。

➢ UARTEnable( )。

➢ UARTParityModeGet( )。

➢ UARTParityModeSet( )。

2）由下列函数使能或禁止 DMA 操作：

➢ UARTDMAEnable( )。

➢ UARTDMADisable( )。

3）由以下函数来实现 UART 的发送和接收：

➢ UARTCharGet( )。

➢ UARTCharGetNonBlocking( )。

➢ UARTCharPut( )。

➢ UARTCharPutNonBlocking( )。

➢ UARTBreakCtl( )。

➢ UARTCharsAvail( )。
➢ UARTSpaceAvail( )。

4）UART 中断由以下函数实现：

➢ UARTIntClear( )。
➢ UARTIntDisable( )。
➢ UARTIntEnable( )。
➢ UARTIntRegister( )。
➢ UARTIntStatus( )。
➢ UARTIntUnregister( )。

3.2.2 UART 的基本操作

（1）初始化 UART

1）使能 UART 外设，如：

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
```

2）设置 RX/TX 引脚作为 UART 引脚，如：

```
GPIOPinConfigure(GPIO_PA0_U0RX);
GPIOPinConfigure(GPIO_PA1_U0TX);
GPIOPinTypeUART(GPIO_PORTA_BASE,GPIO_PIN_0 | GPIO_PIN_1);
```

3）配置 UART 参数，如：

```
ROM_UARTConfigSetExpClk(UART0_BASE,ROM_SysCtlClockGet(),115200,
                        UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                        UART_CONFIG_PAR_NONE));
```

4）配置 UART 的其他特性（如中断、FIFO）。

（2）发送/接收一个字符

1）单寄存器用于发送/接收。

2）在驱动库中的阻塞/非阻塞函数，如：

```
UARTCharPut(UART0_BASE,'a');
newchar = UARTCharGet(UART0_BASE);
UARTCharPutNonBlocking(UART0_BASE,'a');
newchar = UARTCharGetNonBlocking(UART0_BASE);
```



3.3 例程

下面将以 TI 提供的例程为例来介绍 UART 固件库的使用、编程及 IAR 开发软件的使用方法（该例程是自收－自发，不需要外接其他硬件）。

1） uart_echo.c 程序介绍。

```
//*****************************************************************
```

```
//! 文件名:uart_echo.c
//! 来源:TI 例程
//! 功能描述:
//! 使用 UART 实现文字回显
//! UART 配置为:波特率 = 115,200、8 个数据位 - 无校验位 - 1 个停止位
//! 通过虚拟串口(USB 口),将从 UART 接收到的所有字符回送到电脑屏幕上
//! 练习 IAR 开发软件的使用
//! 该例程为版本号为 2.0.1.11577 固件包中的一个例程(用于 EK - TM4C123GXL 开发板)
//*****************************************************************
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "driverlib/debug.h"
#include "driverlib/fpu.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
//*****************************************************************
//当驱动程序库遇到错误时调用的错误处理程序
//*****************************************************************
#ifdef DEBUG
void
__error__(char *pcFilename,uint32_t ui32Line)
{
}
#endif

//*****************************************************************
//UART 中断处理器
//*****************************************************************
void
UARTIntHandler(void)
{
    uint32_t ui32Status;
    //
    //获取中断状态
    //
    ui32Status = ROM_UARTIntStatus(UART0_BASE,true);
    //
    //清除有效中断
    //
    ROM_UARTIntClear(UART0_BASE,ui32Status);
    //
    //判断接收 FIFO 中是否有字符存在
    //
```

```
    while(ROM_UARTCharsAvail(UART0_BASE))
    {
        //
        //从 UART 读取下一个字符,并将其写回到 UART 上
        //
        ROM_UARTCharPutNonBlocking(UART0_BASE,
                                   ROM_UARTCharGetNonBlocking(UART0_BASE));
        //
        //以闪烁的 LED 灯用来显示字符转移正在发生
        //
        GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2,GPIO_PIN_2);
        //
        //延迟 1ms(每个 SysCtlDelay 大约是 3 个时钟)
        //
        SysCtlDelay(SysCtlClockGet()/(1000 * 3));
        //
        //关闭 LED
        //
        GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2,0);
    }
}
//*****************************************************************
//向 UART 发送一个字符串
//*****************************************************************
void
UARTSend(const uint8_t *pui8Buffer,uint32_t ui32Count)
{
    //
    //等待多个字符发送完成
    //
    while(ui32Count--)
    {
        //
        //写下一个字符到 UART
        //
        ROM_UARTCharPutNonBlocking(UART0_BASE,*pui8Buffer++);
    }
}
//*****************************************************************
//该例程将示范如何向 UART 发送一个字符串数据
//*****************************************************************
int
main(void)
{
    //
    //使能扩展(lazy)堆栈的中断处理程序
    //这将允许在中断服务程序间使用浮点指令,但需花费额外的堆栈空间
    //
    ROM_FPUEnable();
```

```
ROM_FPULazyStackingEnable( );
//
//设置直接从晶体运行的时钟
//
ROM_SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_16MHZ);
//
//使能板载用于 LED 灯的 GPIO 端口
//
ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
//
//使能 PF2 引脚来点亮蓝色 LED 灯
//
ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,GPIO_PIN_2);
//
//使能所使用的外设
//
ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
//
//使能处理器中断
//
ROM_IntMasterEnable( );
//
//将 GPIO A0 与 A1 引脚设置为 UART 功能
//
GPIOPinConfigure(GPIO_PA0_U0RX);
GPIOPinConfigure(GPIO_PA1_U0TX);
ROM_GPIOPinTypeUART(GPIO_PORTA_BASE,GPIO_PIN_0 | GPIO_PIN_1);
//
//配置 UART :波特率为 115,200、8 个数据位 - 无校验位 - 1 个停止位(8 - N - 1)
//
ROM_UARTConfigSetExpClk(UART0_BASE,ROM_SysCtlClockGet( ),115200,
                        (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                        UART_CONFIG_PAR_NONE));
//
//使能 UART 中断
//
ROM_IntEnable(INT_UART0);
ROM_UARTIntEnable(UART0_BASE,UART_INT_RX | UART_INT_RT);
//
//提示要输入的文字
//
UARTSend((uint8_t *)"\033[2JEnter text: ",16);
//
//无限循环,以便通过 UART 回显数据
//
while(1)
```

```
            }
        }
    }
```

2）在 C:\ti\TivaWare_C_Series－2. 0. 1. 11577\examples\boards\EK－TM4C123GXL 目录下，导入 IAR EK－TM4C123GXL 工作空间，如图 3-3 所示。

3）如果是首次使用 IAR，一般需要对驱动函数库进行一次编译，如图 3-4 所示。

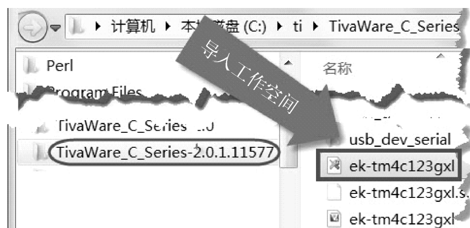


图 3-3　导入 IAREK－TM4C123GXL 工作空间

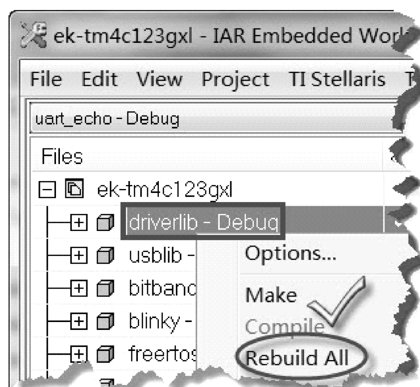


图 3-4　编译驱动函数库

4）激活并编译 uart_echo 工程，如图 3-5 所示。

5）在自建 UART 类工程时，需添加符号定义到 c/c＋＋Compiler 选项中，如图 3-6 所示。

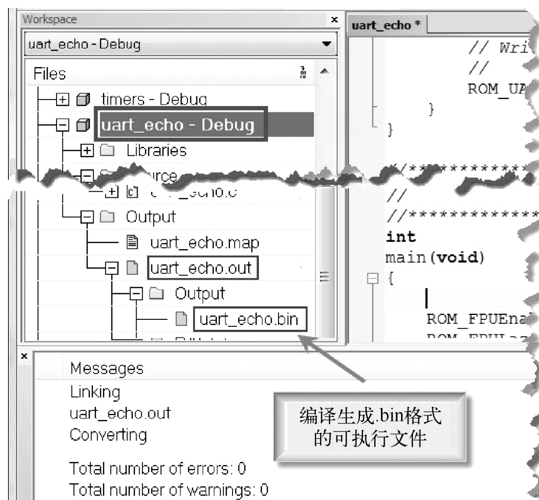


图 3-5　uart_echo 工程的编译结果

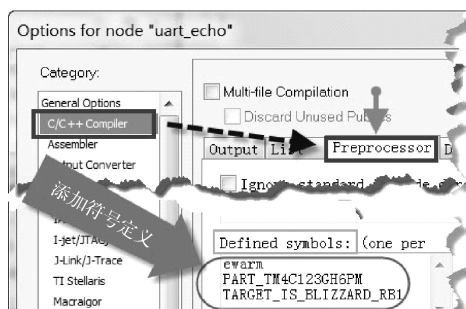


图 3-6　在创建与 UART 有关的工程时需添加符号定义

6）在程序中如果有中断处理函数，需将其添加到中断向量表中。如本例程中的 UARTIntHandler( )函数，首先需在 startup_ewarm. c 中声明为外部函数，然后将中断处理函数添加到该文件的中断向量表中，如图 3-7 所示。

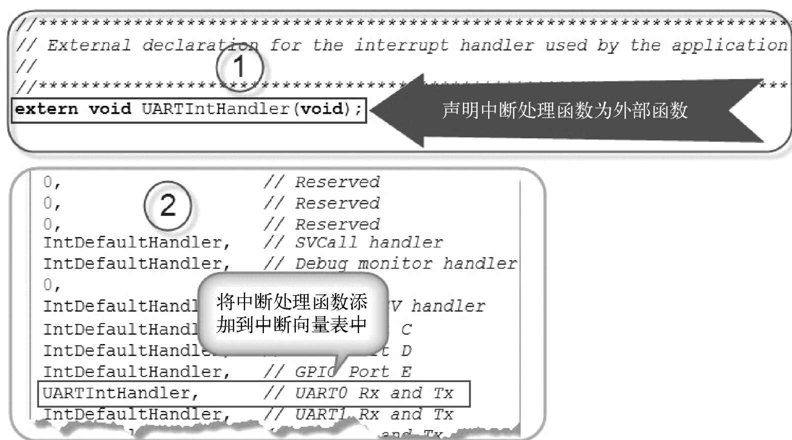


图 3-7　将中断处理函数添加到中断向量表中的方法

7）测试。

① 单击工具栏中的按钮，将编译生成的 .bin 文件下载到 EK-TM4C123GXL 板中，然后单击工具栏中的按钮，让程序在 EK-TM4C123GXL 板中全速运行。

② 打开 PuTTY（有关 PuTTY 的配置请参看第 1 章相关内容），然后按开发板上的复位按钮，显示的程序测试结果如图 3-8 所示。



图 3-8　在 PuTTY 中显示"提示输入文本"

③ 再在图 3-8 绿色光标处，从键盘输入"It is a test for UART!!"文本，并且在输入文本时蓝色的 LED 灯不断闪烁，程序在 EK-TM4C123GXL 板中的运行结果如图 3-9 所示。

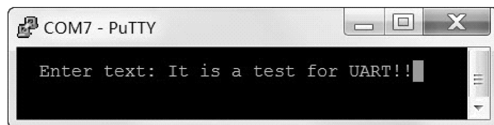


图 3-9　回送程序的测试结果

从图 3-9 中可以看到，通过 UART 接收到的文本"It is a test for UART!!"被回送到 UART 上显示了出来，这就验证了回送程序的正确性。

④ 也可以用串口助手来调试 UART 回送例程，其操作步骤如下：

首先将 UARTSend((uint8_t *)"\033[J2Enter text: ",18) 改为 UARTSend((uint8_t *)"输入文本: ",18)；然后全速运行程序，让 UART 向串口助手发送"输入文本:"文本；最后在串口助手的"发送区"向 UART 发送"This is a test!"文本，同时可观察到蓝色 LED 灯的闪烁，如图 3-10 所示。

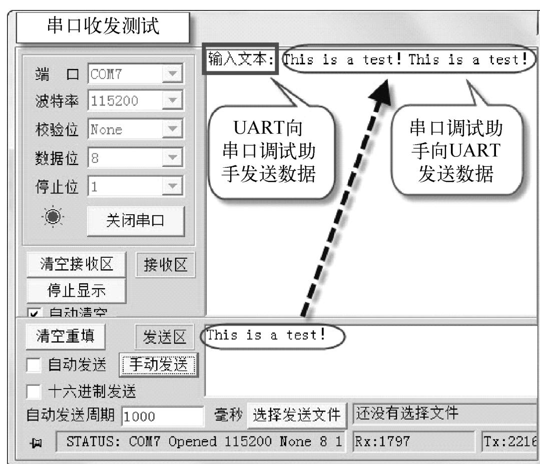


图 3-10　使用串口助手测试 ARUT 回送程序

有兴趣的读者还可以在语句“while(ROM_UARTCharsAvail(UART0_BASE))”处放置一个断点，然后采用单步运行的方式，观察串口助手向 UART 发送文本数据的情况。

⑤ 没有 EK-TM4C123GXL 板子的读者，可用 Proteus 对 dk-lm3s301 软件包中的 uart_echo 工程进行测试，请参考配套资源中的第 3 章内容。在 Proteus 中的测试结果如图 3-11 所示，其方法可参考第 2 章中有关 Proteus 的介绍。

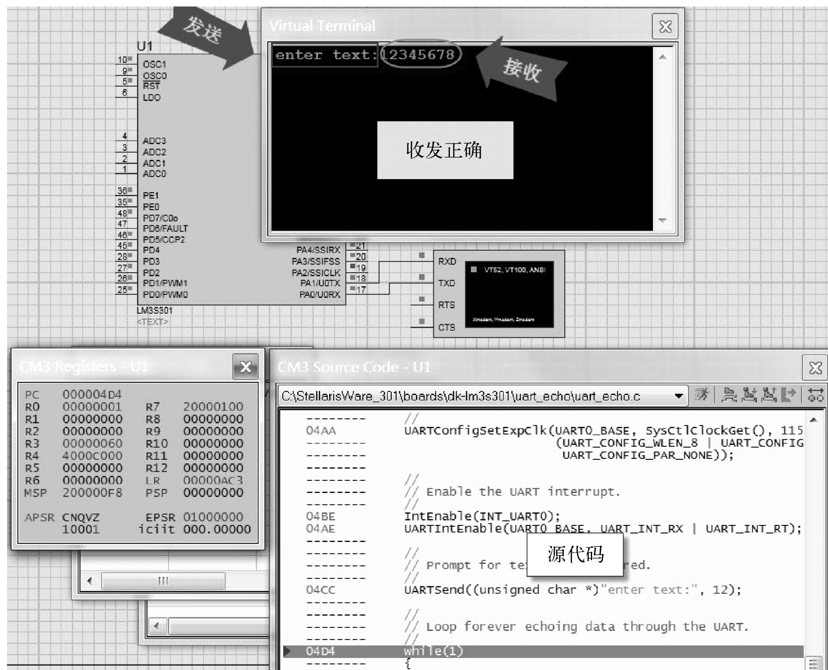


图 3-11　uart_echo 工程在 Proteus 中的测试结果

第4章

模数转换器（ADC）

模数转换器（Analog－to－Digital Converter，ADC）为将连续模拟电压转换为离散数字量的外设。它包含两个完全相同的转换器模块，共用 12 个输入通道。

TM4C123GH6PM ADC 模块具有 12 位的转换精度，支持 12 个输入通道和一个内部温度传感器。每个 ADC 模块都包含 4 个可编程的序列发生器，无须控制器干预就可对多个模拟输入源进行采样。每个采样序列发生器均可提供灵活的编程完全可配置的输入源、触发事件、中断的产生与序列发生器的优先级等。此外，转换后的值还可选择转移到一个数字比较器模块中，且每个 ADC 模块均可提供 8 个数字比较器。每路数字比较器可将 ADC 转换后的值与 2 个由用户定义的值进行评估，以确定信号的工作范围。ADC0 和 ADC1 既可采用独立的触发源，也可采用相同的触发源，以及既可采用相同输入信号，也可采用不同的模拟输入。移相器可由指定的相位角来延迟启动采样，则当同时使用 ADC 模块时，它的采样点既可配置成同相方式，也可配置为互成一定的相角。

ADC API 提供了一组用于操作 ADC 的函数。其固件库函数可用来配置采样序列、读取采样值、配置采样序列的中断处理程序与处理中断屏蔽/清除等。该驱动包含在 driverlib/adc. c 中，driverlib/adc. h 含有应用程序使用的 API 的定义。

本章主要内容：

- ADC 模块简介
- ADC 固件库函数（见附录 B）
- 例程



4.1 ADC 模块

4.1.1 ADC 特点

Tiva TM4C123GH6PM 微控制器提供了两个 ADC 模块（如图 4-1 所示），每个都具有以

下功能：

1）12 个共用模拟输入通道。

2）12 位高精度 ADC。

3）可配置为单端和差分输入。

4）片上内部温度传感器。

5）最大采样率为一百万次采样/秒。

6）可选的可编程采样周期使相移从 22.5°到 337.5°。

7）4 个可编程的采样转换序列发生器，其长度为 1 到 8 个单元，且带有相应的转换结果 FIFO。

8）灵活的触发控制：

① 控制器（软件）。

② 定时器。

③ 模拟比较器。

④ GPIO。

9）硬件可对多达 64 个采样平均。

10）数字比较器模块提供 8 个数字比较器。

11）使用 VDDA 和 GNDA 作为转换器的基准电压。

12）模拟电源与数字电源，以及模拟地和数字地是彼此分开的。

13）用微型直接内存访问控制器（μDMA）进行高速传输：

① 每个采样序列的专用通道。

② ADC 模块的 DMA 使用突发请求。

4.1.2 ADC 模块框图

TM4C123GH6PM 微控制器包含两个相同的模拟/数字转换器（ADC）模块。这两个模块（ADC0 和 ADC1）共享相同的 12 个模拟输入通道。每个 ADC 模块独立运作，因此可以执行不同的采样序列，在任何时间均可对任一模拟输入通道进行采样，并产生不同的中断和触发事件。如图 4-1 所示。

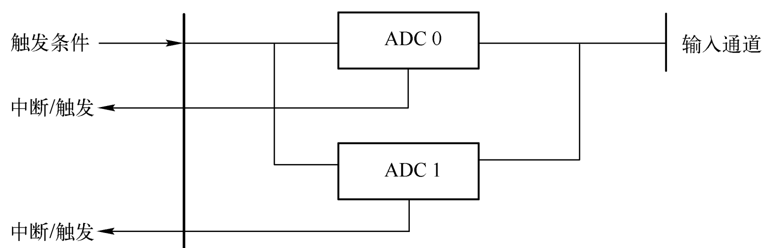
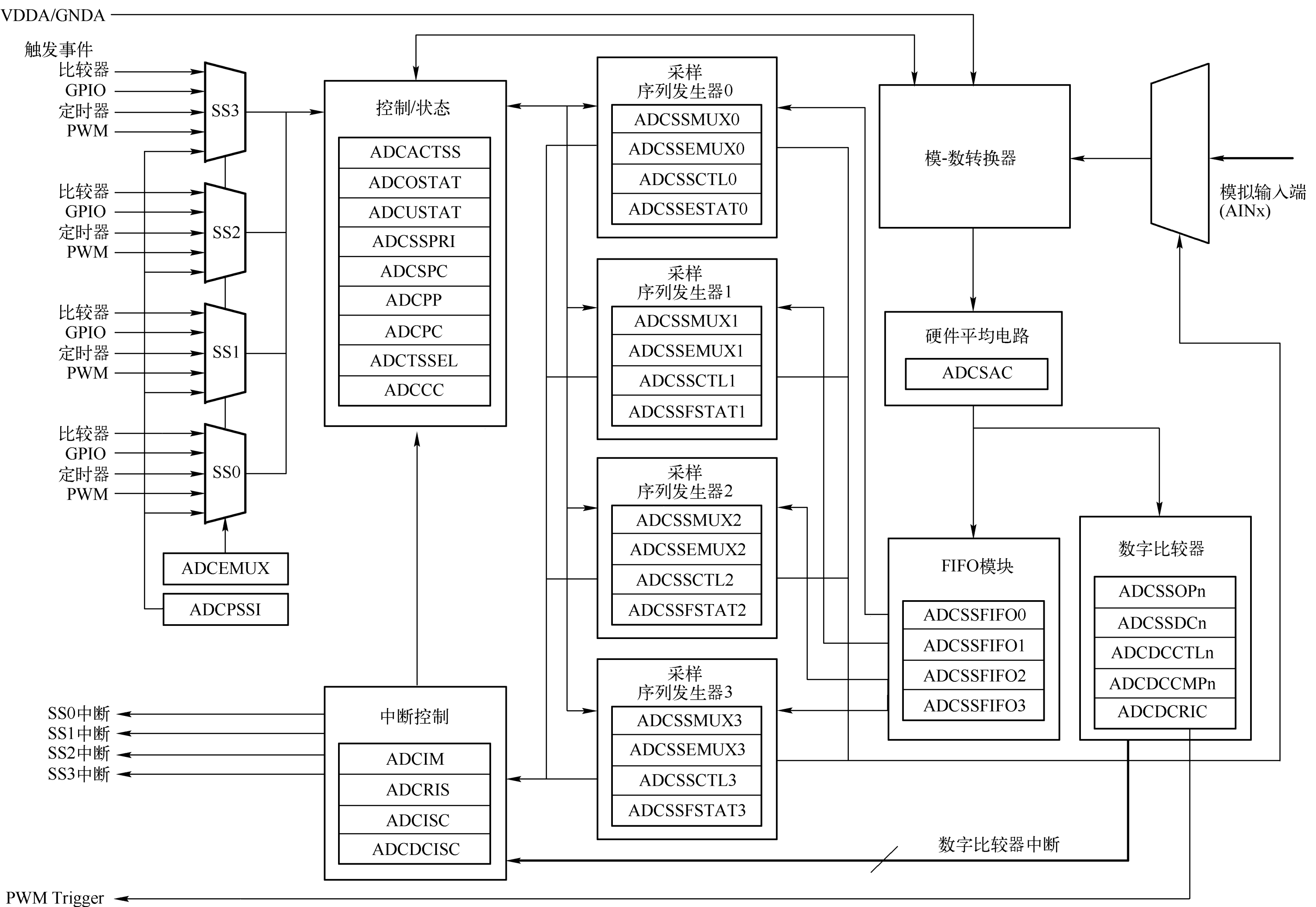


图 4-1　两个 ADC 模块的实现框图

ADC 的模块框图如图 4-2 所示。



VDDA/GNDA
触发事件
比较器
GPIO
定时器
PWM
SS3
比较器
GPIO
定时器
PWM
SS2
比较器
GPIO
定时器
PWM
SS1
比较器
GPIO
定时器
PWM
SS0
ADCEMUX
ADCPSSI
控制/状态
ADCACTSS
ADCOSTAT
ADCUSTAT
ADCSSPRI
ADCSPC
ADCPP
ADCPC
ADCTSSEL
ADCCC
采样
序列发生器0
ADCSSMUX0
ADCSSEMUX0
ADCSSCTL0
ADCSSESTAT0
采样
序列发生器1
ADCSSMUX1
ADCSSEMUX1
ADCSSCTL1
ADCSSFSTAT1
采样
序列发生器2
ADCSSMUX2
ADCSSEMUX2
ADCSSCTL2
ADCSSFSTAT2
采样
序列发生器3
ADCSSMUX3
ADCSSEMUX3
ADCSSCTL3
ADCSSFSTAT3
模-数转换器
模拟输入端
(AINx)
硬件平均电路
ADCSAC
FIFO模块
ADCSSFIFO0
ADCSSFIFO1
ADCSSFIFO2
ADCSSFIFO3
数字比较器
ADCSSOPn
ADCSSDCn
ADCDCCTLn
ADCDCCMPn
ADCDCRIC
SS0中断
SS1中断
SS2中断
SS3中断
中断控制
ADCIM
ADCRIS
ADCISC
ADCDCISC
数字比较器中断
PWM Trigger

图4-2　ADC模块框图

4.1.3 信号描述

ADC 模块的外部信号及其功能描述见表 4-1。

表 4-1 ADC 信号（64LQFP）

引脚名称	引脚号	引脚复用/引脚赋值	引脚类型	缓冲区类型	描述
AIN0	6	PE3	I	模拟	模数转换器输入 0
AIN1	7	PE2	I	模拟	模数转换器输入 1
AIN2	8	PE1	I	模拟	模数转换器输入 2
AIN3	9	PE0	I	模拟	模数转换器输入 3
AIN4	64	PD3	I	模拟	模数转换器输入 4
AIN5	63	PD2	I	模拟	模数转换器输入 5
AIN6	62	PD1	I	模拟	模数转换器输入 6
AIN7	61	PD0	I	模拟	模数转换器输入 7
AIN8	60	PE5	I	模拟	模数转换器输入 8
AIN9	59	PE4	I	模拟	模数转换器输入 9
AIN10	58	PB4	I	模拟	模数转换器输入 10
AIN11	57	PB5	I	模拟	模数转换器输入 11

4.1.4 功能简介

1. 采样序列发生器

采样控制和数据采集都是由采样序列发生器（Sample Sequencer）处理的。所有序列发生器都是相同的，唯一的差别在于其能够捕获的样本数量与 FIFO 的深度（见表 4-2）。捕捉到的每个采样都要存入 FIFO 中。在此实现中，每个 FIFO 入口均为一个 32 位的字，其低 12 位包含的是转换结果。

表 4-2 序列发生器的采样数和 FIFO 深度

序列发生器	采 样 数 量	FIFO 深度
SS3	1	1
SS2	4	4
SS1	4	4
SS0	8	8

对于指定的采样序列，若以 n 代表其序号，则采样序列 n 中的每个采样动作分别以 ADC 采样序列输入多路复用器选择寄存器（ADCSSMUXn）与 ADC 采样序列控制寄存器（ADCSSCTLn）中的 1 个半字节来定义。该 ADCSSMUXn 用于选择输入引脚，而 ADCSSCTLn 包含采样控制位，这些控制位分别与参数（例如，温度传感器的选择、中断使能、序列末端和差分输入模式等）对应。采样序列发生器可以通过置位 ADC 来激活采样序列发生器寄存器（ADCACTSS）中的相应 ASENn 位进行使能，且可在使能之前进行配置。软件可通过置位 ADC 处理器采样序列使能寄存器（ADCPSSI）中的 SSn 位来使能采样。此外，通过配

置 ADCPSSI 寄存器的 GSYNC 和 SYNCWAIT 位可同时使能多个 ADC 模块的多个采样序列。

2. 模块控制

在控制逻辑单元中，除采样序列发生器之外的其余部分将负责执行以下任务：

① 产生中断。

② DMA 操作。

③ 采样序列按优先级执行。

④ 配置触发事件。

⑤ 配置比较器。

⑥ 采样相位控制。

⑦ 模块时钟。

大多数的 ADC 控制逻辑都是在 16 MHz 的 ADC 时钟速率下运行的。当系统 XTAL 选定 PLL 时，硬件将自动配置内部的 ADC 分频系数，使之在 16 MHz 频率进行操作。

（1）中断

采样序列发生器和数字比较器的寄存器配置可决定哪些事件会产生原始中断，但对于中断是否真的发送给中断控制器并没有控制权。ADC 模块是否产生中断信号是由 ADC 中断屏蔽寄存器（ADCIM）中的 MASK 位决定的。中断状态可从两个位置查询：其一，ADC 原始中断状态寄存器（ADCRIS）显示各个中断信号的原始状态；其二，ADC 中断与清除寄存器（ADCISC）显示经 ADCIM 寄存器使能后的实际中断状态。通过向 ADCISC 对应的 IN 位写 1 来清除中断。注意，数字比较器中断不是通过本寄存器清除的，而是通过向 ADC 数字比较器中断状态与清除寄存器（ADCDCISC）的对应位写 1 来清除的。

（2）DMA 操作

如果使用 DMA，则每个采样序列发生器都能独立工作，无须微控制器干预或重置就可传输数据来提高效率。每个采样序列发生器均可向 μDMA 控制器中的相关专用通道发送请求，但 ADC 不支持单次传输请求。当采样序列的中断标志位置位时（ADCSSCTLn 寄存器的 IE 位置位）将发出突发传输请求。而 μDMA 传输的仲裁大小必须是 2 的整数幂，且 ADDSSCTLn 寄存器的相关 IE 位必须置位。

（3）优先级

当同时出现多个采样事件（触发条件）时，可按 ADC 采样序列器优先级寄存器（ADCSSPRI）所定义的优先级顺序依次执行。优先级的有效值为 0～3，其中 0 代表最高优先级、3 代表最低优先级。如果多个激活的采样序列具有相同的优先级，将导致转换结果数据不连续，因此软件必须确保当前激活的所有采样序列各自具有唯一的优先级。

（4）采样事件

每个采样序列发生器的采样触发条件均可通过 ADC 事件复用选择寄存器（ADCEMUX）来定义。触发事件源包括微控制器触发（默认值）、模拟比较器触发、由 GPIO ADC 控制寄存器（GPIOADCCTL）指定的 GPIO 外部信号触发、通用定时器触发和连续采样触发。可通过将 ADC 处理器采样序列使能寄存器（ADCPSSI）中的 SSx 位置位来使能采样序列。在配置连续采样触发条件时务必慎重，若某个采样序列的优先级过高，可能导致其他低优先级采样序列始终无法运行。一般将使用连续采样模式的采样序列器设置为最低优先级，当输入口上的电压达到某一定值时，连续采样可与数字比较器配合使用来产生中断。

（5）采样相位控制

ADC0 和 ADC1 可单独采用不同的触发源，也可采用相同的触发源；既可单独采用不同的模拟输入端，也可采用同一模拟输入端。若两个转换器以相同的采样率工作，其采样点既可以配置为同相，也可以配置为相互错开一定的相角。采样点延后的相位在 0°到 337.5°之间以 22.5°递增，并通过 ADC 采样相位控制寄存器（ADCSPC）进行设置，如图 4-3 所示。

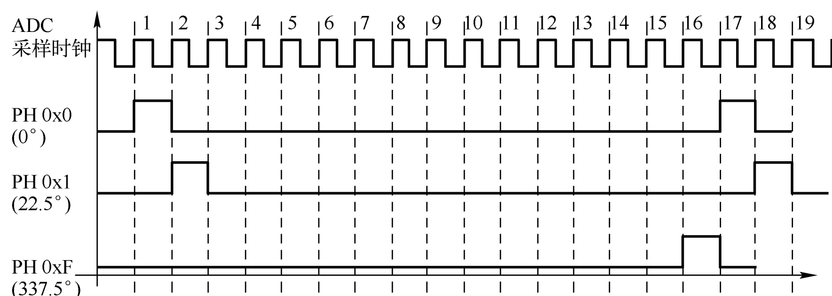


图 4-3　采样相位

（6）模块时钟

该模块的 16 MHz 时钟可由 PLL 分频输出、PIOSC 或连接到 MOSC 的外部时钟源获得。当 PLL 处于工作状态时，ADC 时钟默认值为 PLL÷25。而 PIOSC 可用于使用 ADC 时钟配置寄存器（ADCCC）的模块时钟。若将 PIOSC 作为 ADC 的时钟，首先给 PLL 上电，然后在 ADCCC 寄存器中的 CS 位字段使能 PIOSC，最后禁止 PLL。当 PLL 被旁路时，连接到 MOSC 模块的时钟源必须为 16 MHz，否则只能将 PIOSC 作为时钟源。要使用 MOSC 时钟的 ADC，首先给 PLL 上电，然后使能 ADC 模块时钟，再后禁用 PLL 并切换到 MOSC 系统时钟。如果 PIOSC 是 ADC 模块的时钟源，则 ADC 模块可以继续在深度睡眠模式下运行。

系统时钟必须≥ADC 模块的时钟。使所有 ADC 模块共享相同的时钟源有助于在多个转换设备间同步数据采样。

3. 硬件采样平均电路

使能硬件采样平均电路虽可获得更高的精度，但其代价是吞吐率会成比例地降低。硬件采样平均电路最高可将 64 次采样结果累加并计算出平均值，以平均值作为单次采样的数据写入序列发生器 FIFO 的 1 个单元中。由于是算术平均值，则吞吐率与求平均值的采样数目成反比。例如，若取 16 次采样进行平均值计算，那么吞吐率将降为 1/16。

默认硬件采样平均电路是关闭的，转换器捕捉的所有数据将直接传送到序列发生器的 FIFO 中。进行平均计算的硬件由 ADC 采样平均控制寄存器（ADCSAC）进行控制。每个 ADC 模块只有一个平均电路，不论单端输入还是差分输入都将执行相同的求平均值操作。

如图 4-4 所示给出了一个范例。其中，ADCSAC 寄存器设置为 0x2 以进行 4×硬件过采样；IE1 被置位以提供采样队列；当第 2 个平均值储存到 FIFO 后，会产生中断。

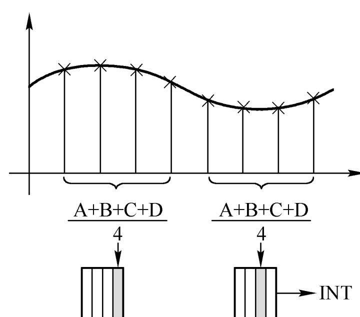


图 4-4　硬件采样平均处理

4. 模数转换器

模数转换器（ADC）模块采用逐次逼近寄存器（Successive Approximation Register（SAR））架构提供一个 12 位、低功耗、高精度的转换值。逐次逼近使用一个开关电容阵列，以实现采样和保持信号的双重功能，并提供 12 位的 DAC 操作。该 ADC 需要 16 MHz 的时钟，这个时钟可以是 PLL 输出的分频值、PIOSC 或连接到 MOSC 的一个 16 MHz 时钟源。ADC 输入等效电路框图如图 4-5 所示。

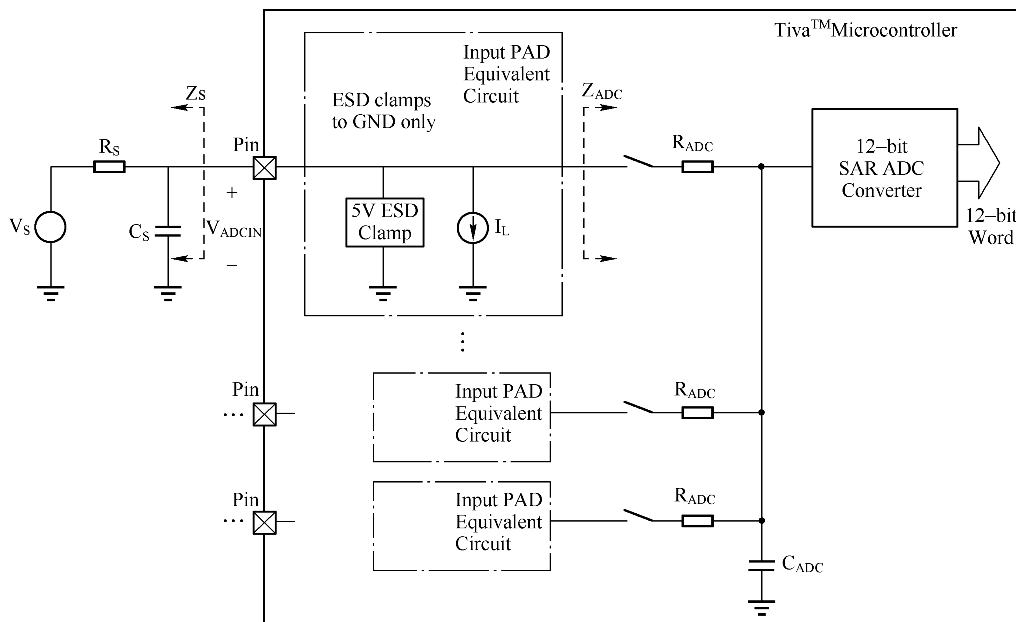


图 4-5　ADC 输入等效电路框图

ADC 模块同时从 3.3 V 模拟电源和 1.2 V 数字电源取电。在不要求 ADC 转换精度时，可以将 ADC 时钟配置为低功耗。从自定义引脚上输入的模拟信号通过特殊的平衡输入通道连接到 ADC，尽量降低输入信号的失真。

（1）基准电压

ADC 使用内部信号 VREFP 和 VREFN 作为基准电压以产生一个选定模拟输入的转换值。VREFP、VREFN 分别连接到 VDDA 和 GNDA，如图 4-6 所示。

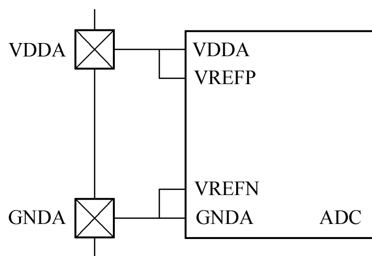


图 4-6　ADC 基准电压

转换值的范围为 0x000 ~ 0xFFF。在单端输入模式，0x000 值对应的电平为 VREFN；而 0xFFF 值对应的电平为 VREFP。此配置的结果可用下列公式计算：

mV per ADC code = (VREFP - VREFN)/4096

尽管模拟输入引脚可以处理超出此范围的电压，但在欠电压及过电压时 A - D 转换结果将会饱和。图 4-7 所示为 ADC 与输入模拟电压的函数关系。

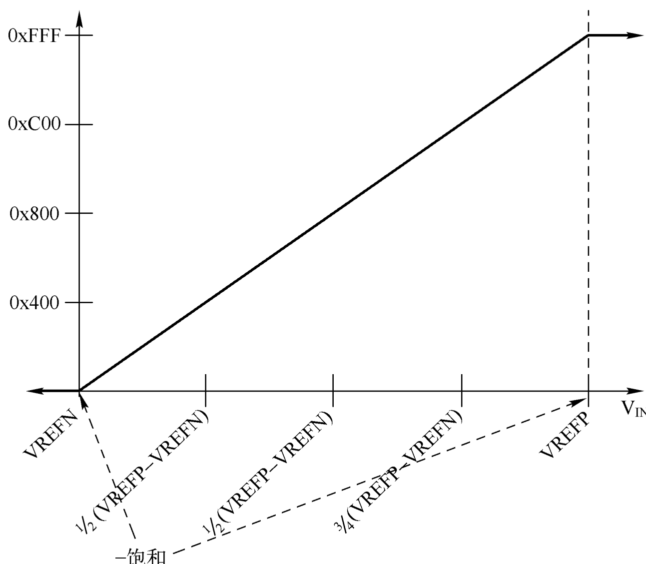


图 4-7　ADC 转换结果

（2）差分采样

除了传统的单端采样，ADC 模块还支持对两个模拟输入通道进行差分采样。差分采样可通过设定 ADCSSCTL0 寄存器中的 Dn 位来使能。若采样序列中某个采样动作配置为差分采样，必须通过 ADCSSMUXn 寄存器来选择输入的差分信号对。差分信号对 0 对模拟输入端 0 和 1 进行采样，差分信号对 1 对模拟输入端 2 和 3 进行采样，依此类推，见表 4-3。

表 4-3　差分采样对

差分信号对	模拟输入端	差分信号对	模拟输入端
0	0 和 1	4	8 和 9
1	2 和 3	5	10 和 11
2	4 和 5	6	12 和 13
3	6 和 7	7	14 和 15

注意： ADC 不支持差分信号对的随机组合，如模拟输入端 0 和模拟输入端 3 无法作为一对差分信号输入。

差分模式下采样电压是奇数通道与偶数通道电压的差值：

差分电压为 ΔV = VIN_EVEN（偶数通道）- VIN_ODD（奇数通道）。

- 如果 ΔV = 0，则转换结果 = 0x800。
- 如果 ΔV > 0，则转换结果 > 0x800（取值范围为 0x800 ~ 0xFFF）。
- 如果 ΔV < 0，则转换结果 < 0x800（取值范围为 0 ~ 0x800）。

对于转换准确度为 $\Delta V \in \{-\Delta VADCV_{REF}, \Delta VADCV_{REF}\}$

分辨率为 mv/ADCCode $= 2 \times \frac{ADCV_{REF+} - ADCV_{REF-}}{4096}$

差分采样如图 4-8 所示。

5. 内部温度传感器

温度传感器的主要作用是当芯片温度过高或过低时向系统给予提示，保障芯片稳定工作。温度传感器没有单独的使能/禁止操作，因为它关系到带隙参考电压的产生，而带隙基准电压不仅提供给 ADC 模块、还需要提供给片内其他所有模拟模块，因此必须始终使能。内部温度传感器提供模拟温度读数以及基准电压 V_{Tsens}（如图 4-9 所示），其大小可以通过下列公式计算：

$$V_{Tsens} = 2.7V - \frac{T(C^0) + 55}{75}$$

$$T(C^0) = 147.5 - \frac{75 \times (VREFP - VRERN) \times ADC_{CODE}}{4096}$$

其中，ADC_{CODE}为 ADC 读数、最大 ADC 电压范围为 VREFP－VREFN。

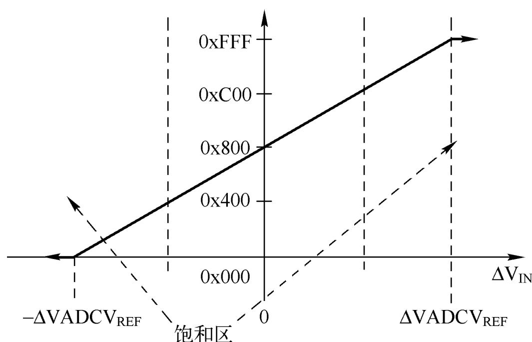


图 4-8　差分采样

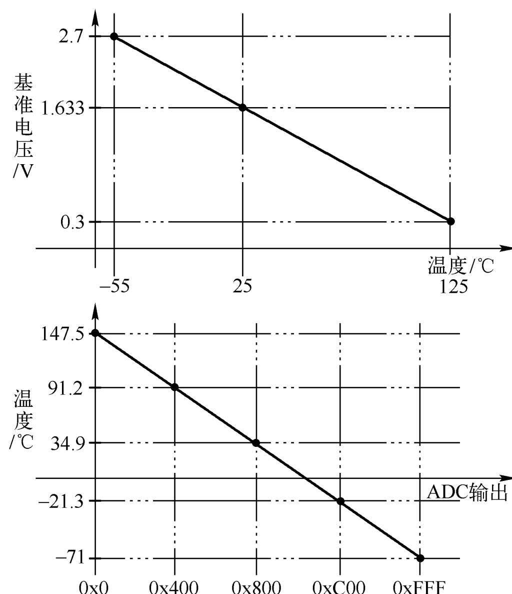


图 4-9　内部温度传感器

6. 数字比较器

ADC 通常用于对外部信号采样并监控其数值的变动，确保其保持在给定的范围内。为了实现此监控过程的自动化、减少所需的处理器开销，每个 ADC 模块提供 8 个数字比较器。ADC 转换结果可直接发送给数字比较器，与用户编程的门限进行比较。门限通过 ADC 数字比较器范围寄存器（ADCDCCMPn）配置。假如被监控的信号超出了容许的范围，则产生一个处理器中断及/或向 PWM 模块发送一个触发事件。用户可将数字比较器的 4 种工作模式（单次触发、持续触发、迟滞单次触发、迟滞持续触发）应用于 3 个相互独立的区域（低频段、中频段、高频段）中。Tiva C ADC 详细的技术文档请参考 TI 的相关文献。



4.2 ADC 固件库函数

模/数转换器的 API 分为三个函数组（如图 4-10 所示）。

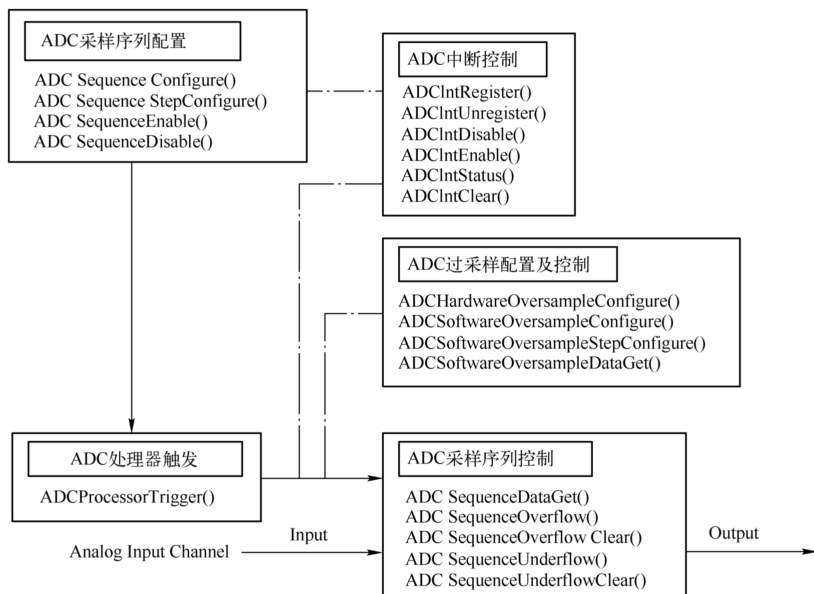


图 4-10　ADC 固件库主要函数及简要使用流程

① 处理采样序列。

② 处理触发。

③ 中断处理。

详细的 ADC 固件库函数功能介绍请参考书后附录 B。



4.3 例程

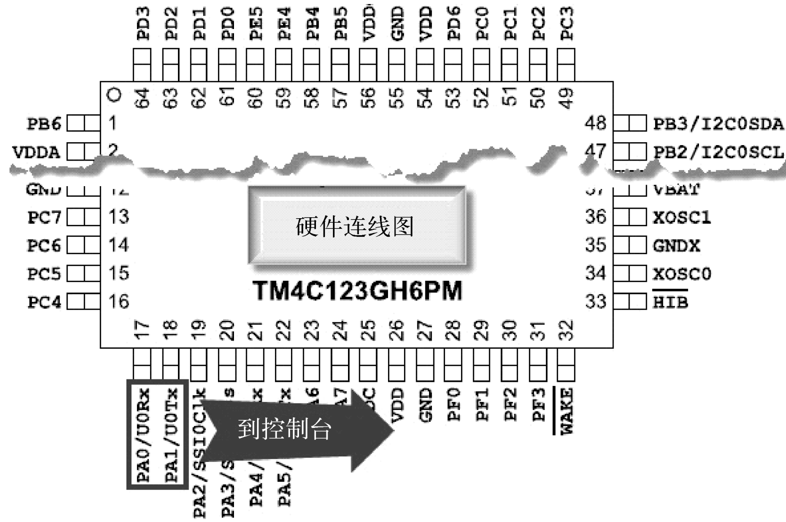
本小节将以 TI 提供的内部温度采集与单端-单输入两个例程为例来介绍 ADC 固件库的使用编程与测试方法。

1. 内部温度传感器例程

1）temperature_sensor. c 程序介绍。

```
//*****************************************************************
//! 文件名:temperature_sensor. c
//! 来源:TI 例程
//! 功能描述:
//! 这个例子介绍了如何通过设置 ADC0 来读取内部温度传感器的值
//! 注:内部温度传感器未经校准,其结果存在误差
//! 本例程将使用下面的外设和 I/O 信号,用户需根据自己的开发板稍作修改:
//!  - ADC0 外设
```

```
//! 用于转换结果显示的 UART 端口：
//!  -UART0 外设
//!  -GPIOA 外设(UART0 引脚)
//!  -UART0RX-PA0
//!  -UART0TX-PA1
//! 本例程为 2.0.1.11577 版固件开发包中的一个示例
/*--------------------------------------------------------------------------
```




```
/*--------------------------------------------------------------------------
//**************************************************************
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "driverlib/adc.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
//**************************************************************
//设置 UART0 用于显示例程运行信息的控制台函数
//**************************************************************
void
InitConsole(void)
{
    //
    //使能用于 UART0 功能的 GPIO 引脚
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    //
    //将复用引脚 A0 和 A1 配置成 UART0 功能
    //
    GPIOPinConfigure(GPIO_PA0_U0RX);
```

```
    GPIOPinConfigure(GPIO_PA1_U0TX);
    //
    //使能 UART0 以便配置时钟
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    //
    //使用内部 16 MHz 振荡器作为 UART 的时钟源
    //
    UARTClockSourceSet(UART0_BASE,UART_CLOCK_PIOSC);
    //
    //选择这些复用引脚为 UART 功能
    //
    GPIOPinTypeUART(GPIO_PORTA_BASE,GPIO_PIN_0 | GPIO_PIN_1);
    //
    //初始化 UART 控制台 I/O
    //
    UARTStdioConfig(0,115200,16000000);
}
//*****************************************************************
//配置 ADC0 的温度传感器的输入为单采样。一旦温度值采样完成,中断标志将被置位,并读取采样数据,然后通过 UART0 将其显示在控制台上
//*****************************************************************
int
main(void)
{
    //
    //这个阵列用于保存从 ADC FIFO 中读取的数据。它必须与使用中的序列发生器中的 FIFO 相同
    //本例中使用序列发生器 3,其 FIFO 深度为 1。如果另一序列发生器使用较深的 FIFO,则数组的大小必须改变
    //
    uint32_t pui32ADC0Value[1];
    //
    //这些变量用于保存转换为摄氏和华氏的温度
    //
    uint32_t ui32TempValueC;
    uint32_t ui32TempValueF;
    //
    //使用 PLL,设置时钟运行在 20 MHz(200 MHz/10)。当使用 ADC 时,则必须使用 PLL 或提供一个 16 MHz 的时钟源
    //
    SysCtlClockSet(SYSCTL_SYSDIV_10 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_16MHZ);
    //
    //调用控制台函数用于显示消息
    //
    InitConsole();
```

```
//
//在控制台上显示以下设置信息
//
UARTprintf("ADC ->\n");
UARTprintf("  Type: Internal Temperature Sensor\n");
UARTprintf("  Samples: One\n");
UARTprintf("  Update Rate: 250ms\n");
UARTprintf("  Input Pin: Internal temperature sensor\n\n");
//
//使能待使用的 ADC0 外设
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
//
//使能采样序列发生器 3,采用处理器信号触发。当处理器发出一个启动 转换信号时,序列 3 将进行单采样
//每个 ADC 模块有 4 个可编程序列发生器,即序列发生器 0~3。本例程使用了序列发生器 3
//
ADCSequenceConfigure(ADC0_BASE,3,ADC_TRIGGER_PROCESSOR,0);
//
//配置序列 3 的步进 0(step 0)。当采样进行时,采样温度传感器(ADC_CTL_TS)和配置中断标志(ADC_CTL_IE)将被设置。告诉 ADC 逻辑,这是序列发生器 3 最后一次转换(ADC_CTL_END)
//序列 3 仅具有一个可编程的 step,而序列 1 和 2 具有 4 个步进(step ),以及序列 0 具有 8 个可编程的步长
//因为仅采用序列 3 进行单次转换,只需配置 step 0 即可
//
ADCSequenceStepConfigure(ADC0_BASE,3,0,ADC_CTL_TS | ADC_CTL_IE |
                         ADC_CTL_END);
//
//使能采样序列发生器 3
//
ADCSequenceEnable(ADC0_BASE,3);
//
//清除中断状态标志,在采样之前需确保中断标志已被清除
//
ADCIntClear(ADC0_BASE,3);
//
//一直采样温度传感器的值,并在控制台上显示出来
//
while(1)
{
    //
    //触发 ADC
    //
    ADCProcessorTrigger(ADC0_BASE,3);
    //
    //等待转换完成
```

```
        //
        while(!ADCIntStatus(ADC0_BASE,3,false))
        {
        }
        //
        //清零 ADC 中断标志
        //
        ADCIntClear(ADC0_BASE,3);
        //
        //读取 ADC 的值
        //
        ADCSequenceDataGet(ADC0_BASE,3,pui32ADC0Value);
        //
        //参考 4.1.4(5)内部温度传感器小节的温度值转换公式
        //其中,VREFP - VRERN = 3V
        //
        ui32TempValueC = ((1475 * 4096) - (2250 * pui32ADC0Value[0]))/40960;
        //
        //计算华氏温度值
        //
        ui32TempValueF = ((ui32TempValueC × 9) + 160)/5;
        //
        //在"串口助手"(PuTTY)上显示摄氏温度值
        //
        UARTprintf("Temperature = %3d * C or %3d * F\r",ui32TempValueC,
                ui32TempValueF);
        //
        //延迟 250 ms(更新速率)
        //
        SysCtlDelay(SysCtlClockGet()/12);
    }
}
```

2）创建 temperature_sensor 工程。其操作步骤如下：

① 创建 temperature_sensor 工程过程，如图 4-11 所示。

② 在 C:\ti\TivaWare_C_Series-2.0.1.11577\examples\peripherals\adc 目录下，导入 temperature_sensor.c 文件。然后按书中代码修改 temperature_sensor.c 程序有关摄氏温度值的计算公式。

③ 配置 temperature_sensor 工程属性，右键单击工程，在 Properties→Build→ARM Compiler→Advanced Options→Predefined Symbols 菜单下添加以下项：

- ccs="ccs"。
- PART_TM4C123GH6PM。
- TARGET_IS_BLIZZARD_RB1。

④ 将 Properties→Build→ARM Linker→Basic Options 菜单下的“Set C system stack size”菜单栏中的内容改为 0x800。本例程规模比较小，改为 512 即可。同时也可以把”Heap size

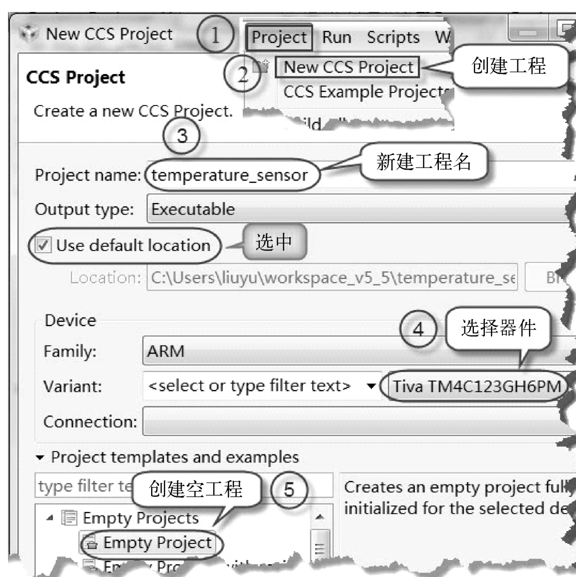


图 4-11　创建 temperature_sensor 工程过程

for C/C ++…” 修改为 200。

3）编译 temperature_sensor 工程，将生成的 .out 文件下载到 LaunchPad（EK－TM4C123GXL）板中，如图 4-12 所示。

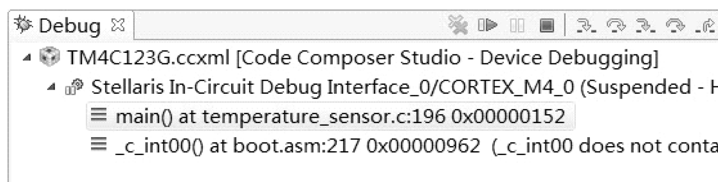


图 4-12　将编译生成的 . out 文件下载到 EK－TM4C123GXL 板中

4）单击图 4-12 中的运行按钮，启动代码在 EK－TM4C123GXL 板中运行，打开 PuTTY 观察到的测试结果如图 4-13 所示。

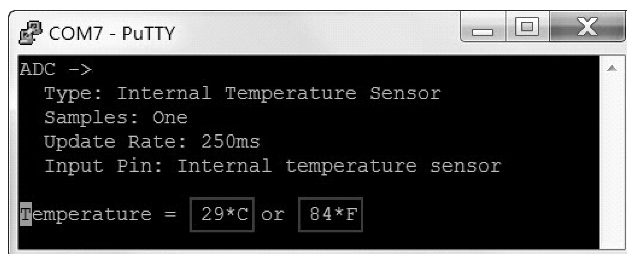


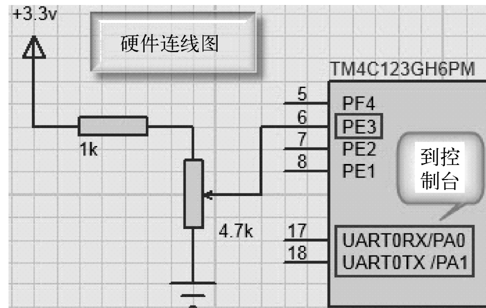
图 4-13　内部温度测试结果

2. 单端－单输入信号 ADC 转换例程

1）single_ended. c 介绍。

```
//*****************************************************************
```

```
//! 文件名:single_ended.c
//! 来源:TI 例程(已修订)
//! 功能描述:
//! 这个例程介绍如何设置 ADC0 作为一个单端输入和执行在 AIN0/PE3 引脚的单采样
//! 此例程使用下列外设和 I/O 信号:
//!  - ADC0 外设
//!  - GPIOE 端口外设(AIN0 引脚)
//!  - AIN0 - PE3
//! 下列配置 UART 信号仅为显示该例程的控制台消息:
//!  - UART0 外设
//!  - GPIO A 端口(UART0 引脚)
//!  - UART0RX - PA0
//!  - UART0TX - PA1
/* ----------------------------------------------------------------
```




```
 ----------------------------------------------------------------*/
// ****************************************************************
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "driverlib/adc.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
// ****************************************************************
//此函数用于设置 UART0 的控制台,以显示该例程运行时的信息
// ****************************************************************
void
InitConsole(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    UARTClockSourceSet(UART0_BASE,UART_CLOCK_PIOSC);
    GPIOPinTypeUART(GPIO_PORTA_BASE,GPIO_PIN_0 | GPIO_PIN_1);
    UARTStdioConfig(0,115200,16000000);
}
```

```
// ***********************************************************
//将 ADC0 配置为单端输入与单采样模式。一旦采样准备就绪,就将设置一个中断标志
//使用轮询的方法读取采样数据,然后通过 UART0 将其显示在控制台上
// ***********************************************************
int
main(void)
{
    //
    //创建保存从 ADC FIFO 中读取数据的阵列及电压值
    //
    uint32_t pui32ADC0Value[1];
    uint32_t ui32Vol_Value;
    //
    //使用 PLL,设置系统时钟运行在 20 MHz(200 MHz/10)
    //
    SysCtlClockSet(SYSCTL_SYSDIV_10 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_16MHZ);
    //
    //调用控制台函数用于显示消息
    //
    InitConsole();
    //
    //在控制台上显示以下设置信息
    //
    UARTprintf("ADC ->\n");
    UARTprintf("  Type: Single Ended\n");
    UARTprintf("  Samples: One\n");
    UARTprintf("  Update Rate: 250ms\n");
    UARTprintf("  Input Pin: AIN0/PE3\n\n");
    //
    //使能待使用的 ADC0 外设
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
    //
    //因为 ADC0 从 AIN0(E3)端口输入,则必须使能 GPIOE 端口
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    //
    //选择 PE3 引脚作为 ADC 输入
    //
    GPIOPinTypeADC(GPIO_PORTE_BASE,GPIO_PIN_3);
    //
    //使能采样序列发生器 3、采用处理器信号触发、采样优先级 0
    //
    ADCSequenceConfigure(ADC0_BASE,3,ADC_TRIGGER_PROCESSOR,0);
    //
    //配置序列 3 的步进 0(step 0)。使用采样通道 0(ADC_CTL_CH0),并在采样完成时配置
    //中断标志(ADC_CTL_IE)。告诉 ADC 逻辑,这是序列 3 的最后一次转换(ADC_CTL_END)
    ADCSequenceStepConfigure(ADC0_BASE,3,0,ADC_CTL_CH0 | ADC_CTL_IE |
```

```
                        ADC_CTL_END);
    //
    //使能采样序列发生器3
    //
    ADCSequenceEnable(ADC0_BASE,3);
    //
    //清除中断状态标志
    //
    ADCIntClear(ADC0_BASE,3);
    //
    //采样 AIN0 的值并在控制台上显示出来
    //
    while(1)
    {
        //
        //触发 ADC 转换
        //
        ADCProcessorTrigger(ADC0_BASE,3);
        //
        //等待转换完成
        //
        while(! ADCIntStatus(ADC0_BASE,3,false))
        {
        }
        ADCIntClear(ADC0_BASE,3);
        ADCSequenceDataGet(ADC0_BASE,3,pui32ADC0Value);
        //
        //将数字电压值转换成电压值(12 位 ADC,3.3v 的数字量为 2^12 =4096)
        //
        ui32Vol_Value =( pui32ADC0Value[0] * 3.3)/4096;
        //
        //在 PuTTY 中显示 AIN0(PE3)的电压值
        //
        UARTprintf("AIN0 = %4d(v)\r",ui32Vol_Value);
        //
        //延迟 250 ms (更新速率)
        //
        SysCtlDelay(SysCtlClockGet()/12);
    }
}
```

2）在 C:\ti\TivaWare_C_Series－2.0.1.11577\examples\peripherals\adc 目录下，导入 single_ended.c 文件，然后按书中内容修改。

3）创建工程、编译等和上例相同，此处省略。

4）测试。

① 读者可以按上述硬件连接图进行，通过调节电位器来改变 ADC 的输入电压，从而得到不同的 ADC 转换值。如果读者觉得外接电位器比较麻烦，也可直接在 EK－TM4C123GXL 板上寻找几个已知的电压值，比如 VDDC（万用表测得该点的电压为 3.27 V）和接地信号来

测试程序的正确性。电压采集点如图 4-14 所示。

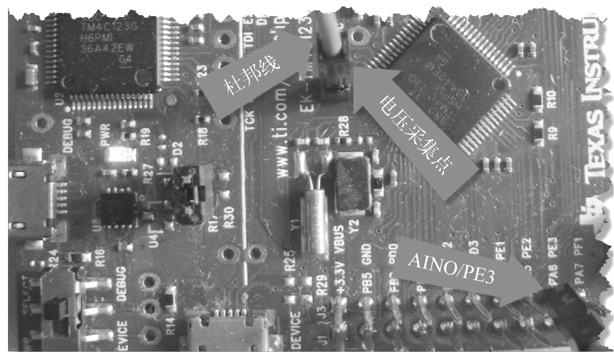


图 4-14　VDDC 电压的采集接线图（用杜邦线将电压值送到 AIN0）

② 将编译生成的 . out 文件下载到 LaunchPad 板中，单击运行按钮，其测试结果如图 4-15 所示。

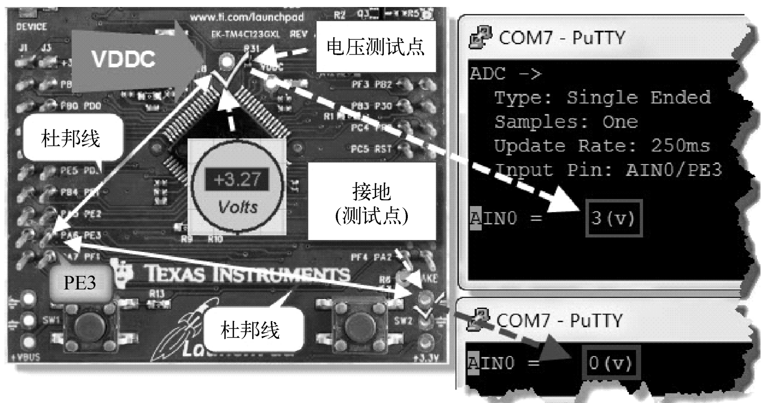


图 4-15　单端 - 单输入信号的 ADC 转换结果（AIN0 =3V or 0v）

从图 4-15 中可以看到，万用表实测的 VDDC =3. 27 V，而 ADC 转换得到数字电压，通过换算后得到的电压值 =3 V，存在误差。下面将进一步验证该误差是由显示控制台程序的显示格式（%d）所致，还是由 ADC 转换的精度不够所造成?

① 在 Expressions 窗口添加 ui32Vol_Value 变量，并按下数据连续更新图标，如图 4-16 所示。

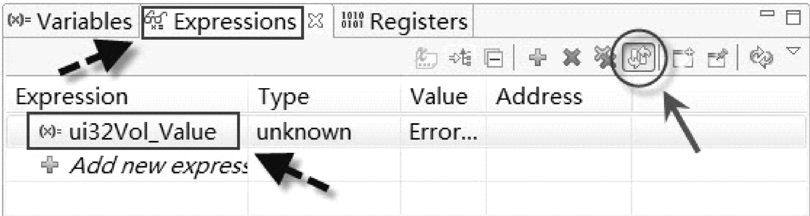


图 4-16　添加 ui32Vol_Value 变量

② 在如图 4-17 所示处设置一个断点。

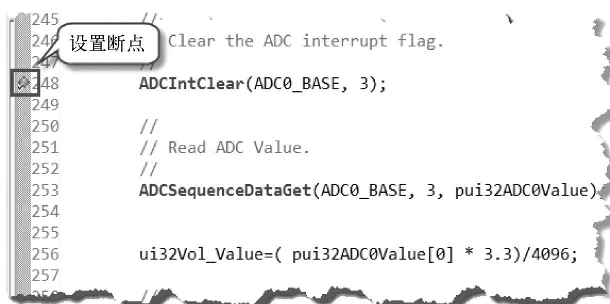


图 4-17　设置断点

③ 连续单击程序运行按钮，直到 Expressions 窗口中变量 ui32Vol_Value 的值不再变化为止，如图 4-18 所示。



图 4-18　ADC 的转换结果

从图 4-18 可以看到，ADC 的转换结果 = 3 V 和前面在 PuTTY 中的显示结果一致，说明误差是由于 TM4C123GH6PM 的片载 ADC 转换器精度不够所致。

在读取某点数据值时，也可以不用按下连续更新图标，而采用通过设置断点属性获得。其步骤如下：

① 右键单击所设断点，在弹出的下拉菜单中选择 Breakpoint Properties 选项。

② 单击 Breakpoint Properties 子菜单，在弹出的下拉菜单中将 Action 选项设置为 Update View，如图 4-19 所示。

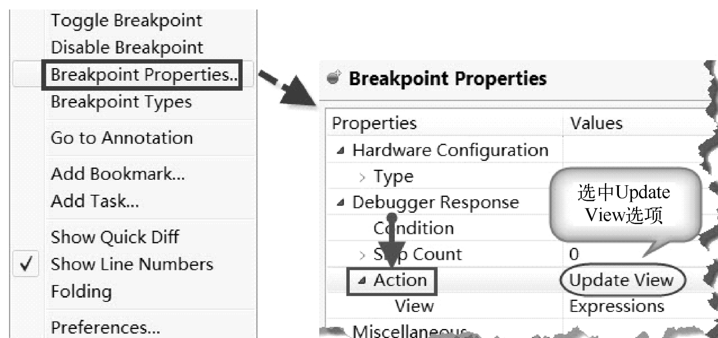


图 4-19　设置数据连续更新模式

讨论：从上面的介绍可以看出，并非是显示方式造成的误差，那只可能是 ADC 转换精度或转换程序存在问题。为了做出这个判断，最佳的办法是：观察 ADC 输出的数字量是否

正确，其步骤如下：

① 将 pui32ADC0Value[0]添加到表达式窗口。

② 添加一个观察 pui32ADC0Value[0]值的断点，如图 4-20 所示。



图 4-20　在图示位置设置一个断点

③ 单击工具栏上的图标让程序运行到断点处并单步执行程序，此时在表达式窗口中观察到的 pui32ADC0Value[0]值为 4095（3.3 V/12 位的 ADC 输出为 4096），说明 ADC 输出精度足够。那么产生上述 ADC 转换结果误差较大的原因是电压换算语句存在问题。即

```
ui32Temp_Value = ( pui32ADC0Value[0] * 3.30)/4096;
```

修改这条语句的步骤如下：

① 添加浮点类型的变量 adcResult，并使能 FPULazy 堆栈，如图 4-21 所示。

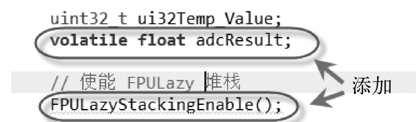


图 4-21　添加变量定义及使能 FPULazy 堆栈

② 将电压换算语句修改为：

```
ui32Temp_Value = pui32ADC0Value[0];
adcResult = (float)(ui32Temp_Value * 3.30/4096);
```

③ 添加一个观察 ADC 转换结果的断点，如图 4-22 所示。

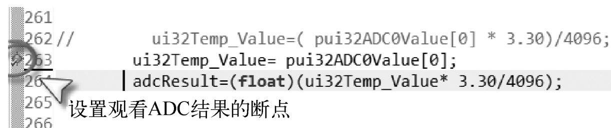


图 4-22　在图示位置设置观察 ADC 转换结果的断点

④ 将 ADC 转换结果 adcResult 添加到表达式窗口中，如图 4-23 所示。

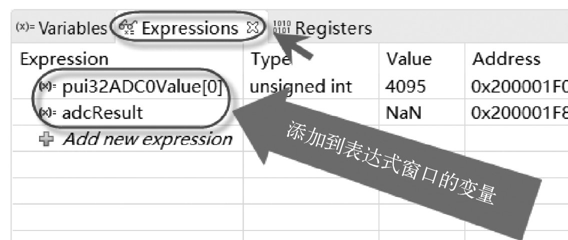


图 4-23　将变量 adcResult 添加到表达式窗口中

⑤ 单击工具栏上的图标让程序运行到断点处并单步执行程序，此时在表达式窗口观察到的 ADC 转换结果如图 4-24 所示。

图 4-24　最后的 ADC 值已过电压换算后的结果

说明：电压换算语句经过上述修改后得出的 3.3V 模拟输入的 ADC 转换结果正确。

第 5 章

通用输入/输出（GPIO）

通用输入/输入端口（General – Purpose Input/Output，GPIO）为微控制器与被控对象间的信息交流通道端口。TM4F123G6HPM 芯片包含 6 个物理 GPIO 模块，每个模块都对应一个单独的 GPIO 端口（即端口 A ~ F）。GPIO 模块支持高达 43 个可编程的输入/输出引脚，并且这些引脚可配置成多种工作模式，如高阻输入、推挽输出与开漏输出等。

GPIO API 提供了一组实现 GPIO 功能的函数，包含在 driverlib/gpio. c 中，而 driverlib/gpio. h 头文件包含了 API 的定义。

本章的主要内容：

- ➢ GPIO 模块
- ➢ GPIO 固件库函数（见附录 C）
- ➢ 例程



5.1 GPIO 模块

5.1.1 GPIO 特点

GPIO 模块具有以下特点：

1）高达 43 个 GPIO 端口，具体取决于外设的配置。

2）高度灵活的复用引脚，可作为 GPIO 或多种外设功能。

3）5 V 容错输入配置。

4）通过高级外设总线（APB）访问端口 A ~ G。

5）具有在 1 个或 2 个时钟周期内快速切换 AHB 端口的能力。

6）可编程控制的 GPIO 中断：

① 屏蔽中断产生。

② 边沿触发（上升沿、下降沿或双沿）。

③ 电平触发（高电平或低电平）。

7）在读和写操作中通过地址线进行位屏蔽。

8）可用于使能 ADC 采样序列或 μDMA 传输。

9）在休眠模式下可保留引脚的状态。

10）可配置数字输入引脚为施密特触发。

11）可编程控制 GPIO 引脚配置：

① 弱上拉或下拉电阻。

② 具有 2 mA、4 mA、8 mA 的数字通信驱动电流；对于需要大电流的应用，多达 4 个引脚可承载 18 mA 电流。

③ 具有 8 mA 驱动的斜率控制。

④ 开漏使能。

⑤ 数字输入使能。

5.1.2 GPIO 模块框图

GPIO 的模拟/数字口模块框图如图 5-1 所示。

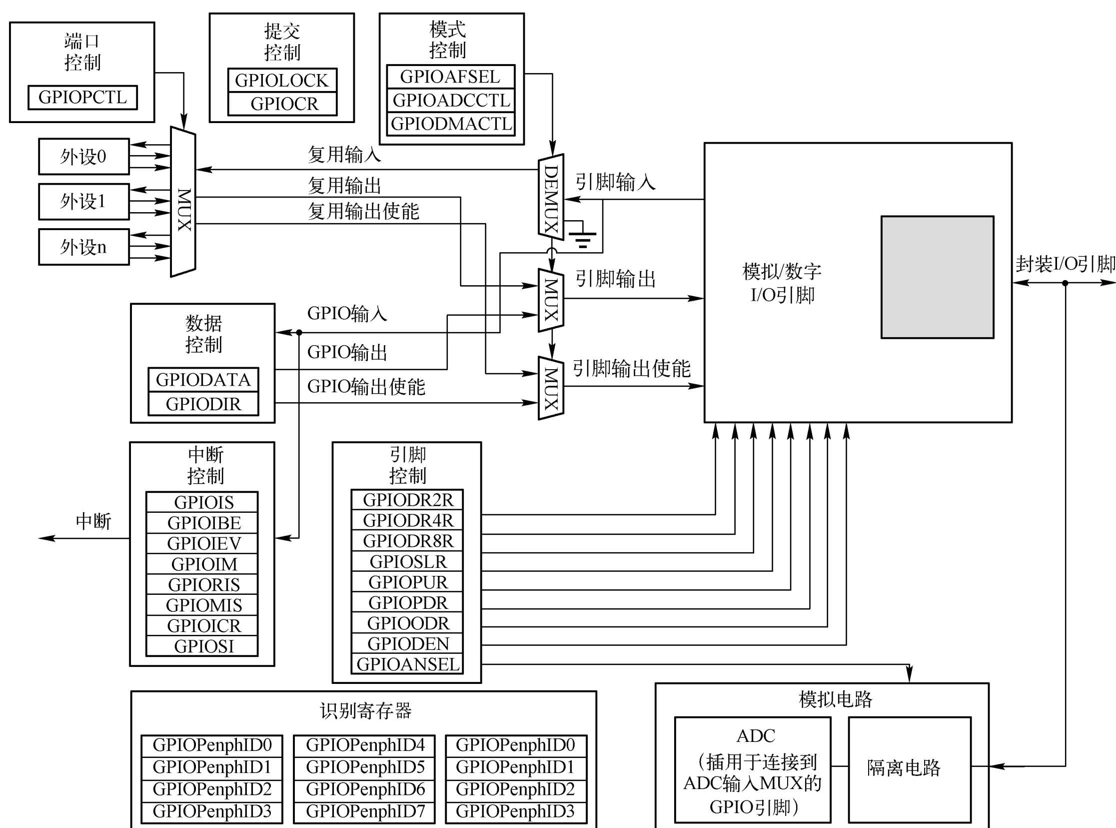


图 5-1　GPIO 模拟/数字口模块框图

5.1.3 功能简介

1. 数据控制

数据控制寄存器允许软件配置 GPIO 的操作模式。当数据寄存器捕获到输入数据时，数据方向寄存器将 GPIO 配置为输入；当数据寄存器通过端口输出数据时，数据方向寄存器将 GPIO 配置为输出。

（1）数据方向操作

GPIO 方向寄存器（GPIODIR）用于将每个独立的引脚配置成输入或输出。当数据方向寄存器中的位被清零时，则配置为输入，使寄存器中的相应位捕获并储存 GPIO 端口的值；当数据方向寄存器中的位被置位时，则配置为输出，数据寄存器中的相应位便可驱动 GPIO 端口。

（2）数据寄存器的操作

为了提高软件的效率，可将地址总线上的［9:2］位用作屏蔽位，对 GPIO 端口的数据寄存器（GPIODATA）中的各个位进行修改。采用这种方式使软件驱动程序仅用一条指令就可修改任意一个 GPIO 引脚，而不会影响其他引脚的状态。该方式与通过“读－修改－写”来操作 GPIO 引脚的传统做法不同。为了实现这种功能，GPIODATA 寄存器覆盖了存储器映射中的 256 个单元。

在进行写入操作时，如果与数据位相关联的地址位被置位，则 GPIODATA 寄存器的值将相应地发生变化，如果地址位被清零，那么数据位将保持不变。

例如，将 0xEB 写入到地址 GPIODATA＋0x98 处，其结果如图 5-2 所示。其中，u 表示写入操作没有改变数据。

在进行读操作时，如果与数据位相关联的地址位被置位，那么就读取数据寄存器中的值；如果与数据位相关联的地址位被清零，则不管数据寄存器中的实际值是什么都读取 0。例如，读取地址 GPIODATA＋0x0C4 处的值，其结果如图 5-3 所示。

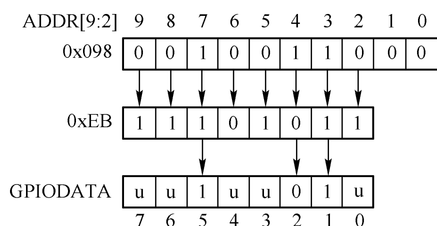


图 5-2　GPIODATA 写入示例

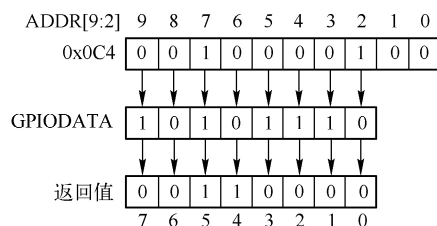


图 5-3　GPIODATA 读取示例

2. 中断控制

每个 GPIO 端口的中断能力都由 7 个寄存器控制。这些寄存器可用于选择中断源、极性与边沿属性。当一个或多个输入产生中断时，只将其中的一个中断输出，发送到供所有 GPIO 端口使用的中断控制器中。对于边沿触发，为使其他中断可用，软件必须清除该中断；对于电平触发，只有保持外部电平的状态，才能使控制器识别中断的发生。

以下三个寄存器用来定义中断触发的边沿或测量：

① GPIO 中断检测寄存器（GPIOIS）。

② GPIO 中断双边沿寄存器（GPIOIBE）。

③ GPIO 中断事件寄存器（GPIOIEV）。

通过 GPIO 中断屏蔽寄存器（GPIOIM）可使能/禁止中断。当产生中断条件时，可在 GPIO 原始中断状态寄存器（GPIORIS）和 GPIO 屏蔽后的中断状态寄存器（GPIOMIS）中观察到中断信号的状态。即 GPIOMIS 寄存器仅显示允许被传送到中断控制器的中断条件。GPIORIS 寄存器则表示 GPIO 引脚满足的中断条件，但不一定发送到控制器中。

注意：往 GPIO 中断清零寄存器（GPIOICR）中的位写 1 可以清除相应的中断。在设置中

断控制寄存器（GPIOIS、GPIOIBE 或 GPIOIEV）时，应该保持中断的屏蔽状态（GPIOIM 清零），如果相应的位没有屏蔽，那么向中断控制寄存器中写入任何值都有可能产生伪中断。

（1）ADC 触发源

任何 GPIO 引脚都可以使用 GPIO ADC 控制寄存器（GPIOADCCTL）将其配置成 ADC 的一个外部触发。如果任一 GPIO 端口被配置成非屏蔽的中断引脚，当该端口产生一个中断时，就会发出一个外部触发信号到 ADC。此时如果 ADC 事件复用选择寄存器（ADCEMUX）被配置为使用外部触发器，那么就会启动 ADC 转换。

（2）μDMA 触发源

任何 GPIO 引脚都可以使用 GPIO DMA 控制寄存器（GPIODMACTL）配置成 μDMA 的一个外部触发源。如果任一 GPIO 端口被配置为非屏蔽的中断引脚，当该端口产生一个中断，就会发出一个外部触发信号到 μDMA。如果根据 GPIO 信号 μDMA 被配置成启动传输，那么就会启动传输。

3. 模式控制

GPIO 引脚既可以被软件控制也可以被硬件控制。大部分的引脚默认为软件控制，此时 GPIODATA 寄存器用来读写相应的引脚。当 GPIO 复用功能选择寄存器（GPIOAFSEL）使能硬件控制时，引脚状态将由它的复用（即外设）功能控制。

更多的引脚复用功能选择由 GPIO 端口控制寄存器（GPIOPCTL）提供，该寄存器可为每个 GPIO 选择其中一个外设功能。注意：如果一个引脚被用作 ADC 输入，那么 GPIOAMSEL 寄存器中相应位必须被置位来禁止模拟隔离电路。

4. 提交（Commit）控制

GPIO 提交控制寄存器提供的保护层可防止对重要硬件外设的意外编程。该功能可保护 4 个 JTAG/SWD 引脚（PC[3:0]）与 NMI 引脚（PD7 和 PF0）。向 GPIO 复用功能选择寄存器（GPIOAFSEL）、GPIO 上拉电阻选择寄存器（GPIOPUR）、GPIO 下拉电阻选择寄存器（GPIOPDR）与 GPIO 数字使能寄存器（GPIODEN）中受保护的位写入数据将不会被确认保存，除非 GPIO 锁定寄存器（GPIOLOCK）没有被锁定，同时 GPIO 提交寄存器（GPIOCR）中的相应位被置位。

5. 引脚（Pad）控制

可根据应用程序的要求用软件来配置 GPIO 引脚。引脚控制寄存器包括 GPIODR2R、GPIODR4R、GPIODR8R、GPIOODR、GPIOPUR、GPIOPDR、GPIOSLR 与 GPIODEN 寄存器。这些寄存器控制着引脚的驱动电流大小、开漏配置、上/下拉电阻的选择、转换率（slew - rate）控制和数字输入使能等。如果在配置为开漏输出的 GPIO 端口上加上 5V 电压，其输出电压将取决于上拉电阻的大小，而没有配置的 GPIO 引脚将输出 5V 电压。

6. 标识

复位时配置的标识寄存器允许软件将模块当作 GPIO 块进行检测和识别。标识寄存器包括 GPIOPeriphID0 ~ GPIOPeriphID7 寄存器与 GPIOPCellID0 ~ GPIOPCellID3 寄存器。

5.1.4 寄存器映射及寄存器描述

1. 寄存器映射

GPIO 中包含的寄存器见表 5-1。每一个 GPIO 端口都可通过两种总线槽访问。传统的一

种称为先进外设总线（APB），向后兼容以前的产品。另外一种是先进高端总线（AHB），它虽和 APB 总线一样拥有相同的寄存器映射，却提供了更好的连续访问性能。

表 5-1　GPIO 寄存器映射

偏 移 量	名　称	类　型	复　位	描　述
0x000	GPIODATA	R/W	0x0000.0000	GPIO 数据寄存器
0x400	GPIODIR	R/W	0x0000.0000	GPIO 方向寄存器
0x404	GPIOIS	R/W	0x0000.0000	GPIO 中断检测寄存器
0x408	GPIOIBE	R/W	0x0000.0000	GPIO 双边沿
0x40C	GPIOIEV	R/W	0x0000.0000	GPIO 中断事件寄存器
0x410	GPIOIM	R/W	0x0000.0000	GPIO 中断屏蔽寄存器
0x414	GPIORIS	RO	0x0000.0000	GPIO 原始中断状寄存器
0x418	GPIOMIS	RO	0x0000.0000	GPIO 中断屏蔽状态寄存器
0x41C	GPIOICR	W1C	0x0000.0000	GPIO 中断清除寄存器
0x420	GPIOAFSEL	R/W	—	GPIO 复用功能选择寄存器
0x500	GPIODR2R	R/W	0x0000.00FF	GPIO 2mA 驱动选择寄存器
0x504	GPIODR4R	R/W	0x0000.0000	GPIO 4mA 驱动选择寄存器
0x508	GPIODR8R	R/W	0x0000.0000	GPIO 8mA 驱动选择寄存器
0x50C	GPIOODR	R/W	0x0000.0000	GPIO 开漏选择寄存器
0x510	GPIOPUR	R/W	—	GPIO 上拉电阻选择寄存器
0x514	GPIOPDR	R/W	0x0000.0000	GPIO 下拉电阻选择寄存器
0x518	GPIOSLR	R/W	0x0000.0000	GPIO 斜率控制选择寄存器
0x51C	GPIODEN	R/W	—	GPIO 数字使能寄存器
0x520	GPIOLOCK	R/W	0x0000.0001	GPIO 锁定寄存器
0x524	GPIOCR	—	—	GPIO 提交寄存器
0x528	GPIOAMSEL	R/W	0x0000.0000	GPIO 模拟选择寄存器
0x52C	GPIOPCTL	R/W	—	GPIO 端口控制寄存器
0x530	GPIOADCCTL	R/W	0x0000.0000	GPIO ADC 控制寄存器
0x534	GPIODMACTL	R/W	0x0000.0000	GPIO DMA 控制寄存器
0xFD0	GPIOPeriphID4	RO	0x0000.0000	GPIO 外设标识寄存器 4
0xFD4	GPIOPeriphID5	RO	0x0000.0000	GPIO 外设标识寄存器 5
0xFD8	GPIOPeriphID6	RO	0x0000.0000	GPIO 外设标识寄存器 6
0xFDC	GPIOPeriphID7	RO	0x0000.0000	GPIO 外设标识寄存器 7
0xFE0	GPIOPeriphID0	RO	0x0000.0061	GPIO 外设标识寄存器 0
0xFE4	GPIOPeriphID1	RO	0x0000.0000	GPIO 外设标识寄存器 1
0xFE8	GPIOPeriphID2	RO	0x0000.0018	GPIO 外设标识寄存器 2
0xFEC	GPIOPeriphID3	RO	0x0000.0001	GPIO 外设标识寄存器 3
0xFF0	GPIOPCellID0	RO	0x0000.000D	GPIOPrimeCell 标识寄存器 0

（续）

偏　移　量	名　　称	类　　型	复　　位	描　　述
0xFF4	GPIOPCellID1	RO	0x0000. 000F0	GPIOPrimeCell 标识寄存器 1
0xFF8	GPIOPCellID2	RO	0x0000. 0005	GPIOPrimeCell 外设标识寄存器 2
0xFFC	GPIOPCellID3	RO	0x0000. 000B1	GPIOPrimeCell 外设标识寄存器 3

下面给出各个 GPIO 端口的基地址和偏移量：

- GPIO 端口 A（APB）：0x4000. 4000。
- GPIO 端口 A（AHB）：0x4005. 8000。
- GPIO 端口 B（APB）：0x4000. 5000。
- GPIO 端口 B（AHB）：0x4005. 9000。
- GPIO 端口 C（APB）：0x4000. 6000。
- GPIO 端口 C（AHB）：0x4005. A000。
- GPIO 端口 D（APB）：0x4000. 7000。
- GPIO 端口 D（AHB）：0x4005. B000。
- GPIO 端口 E（APB）：0x4002. 4000。
- GPIO 端口 E（AHB）：0x4005. C000。
- GPIO 端口 F（APB）：0x4002. 5000。
- GPIO 端口 F（AHB）：0x4005. D000。

2. 几个常用的寄存器描述

（1）GPIO 端口控制寄存器（GPIOPCTL），偏移量 0x52C

GPIOPCTL 寄存器与 GPIOAFSEL 寄存器协作为每个 GPIO 引脚（当使用复用功能模式时）选择具体的外设信号。GPIOAFSEL 寄存器中大部分的位在复位时是清零的，所以大部分 GPIO 引脚默认配置为 GPIO 功能。在 GPIOAFSEL 寄存器中某位被置位时，表示相应的 GPIO 引脚由相关外设控制。GPIOPCTL 寄存器可以为每个 GPIO 引脚选择使用何种外设功能，因此在信号定义时提供了高度的灵活性，如图 5-4 所示。

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	PMC7				PMC6				PMC5				PMC4			
类型	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
复位	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	PMC3				PMC2				PMC1				PMC0			
类型	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
复位	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

图 5-4　端口控制寄存器（GPIOPCTL）

位/字段	名称	类型	复位	描述
31：28	PMC7	R/W	–	端口复用控制器 7 该区域的值控制着端口 7 的配置
27：24	PMC6	R/W	–	端口复用控制器 6

				该区域的值控制着端口 6 的配置
15：12	PMC3	R/W	–	端口复用控制器 3 该区域的值控制着端口 3 的配置
11：8	PMC2	R/W	–	端口复用控制器 2 该区域的值控制着端口 2 的配置
7：4	PMC1	R/W	–	端口复用控制器 1 该区域的值控制着端口 1 的配置
3：0	PMC0	R/W	–	端口复用控制器 0 该区域的值控制着端口 0 的配置

（2）GPIO 数据寄存器（GPIODATA），偏移量 0x000

在软件控制模式中，如果通过 GPIO 方向寄存器（GPIODIR）将各个引脚配置成输出，那么写入 GPIODATA 寄存器的值将被发送到 GPIO 端口引脚上。

为了对 GPIODATA 寄存器执行写入操作，由地址总线位［9:2］产生的相关屏蔽位必须被置位。否则，位的值不会被写入操作改变。

同样，从该寄存器读取的值由从访问数据寄存器的地址处获取的屏蔽位［9:2］的情况来决定。如果地址屏蔽位为 1，那么读取 GPIODATA 中相应位的值；如果地址屏蔽位为 0，那么不管 GPIODATA 中相应位的值是什么，读取得到的结果都是 0。

如果各自的引脚被配置成输出，那么读取 GPIODATA 将返回最后写入的位值；或者当这些引脚被配置成输入时，将返回相应的输入引脚上的值。复位时所有的位都是清零的，如图 5-5 所示。

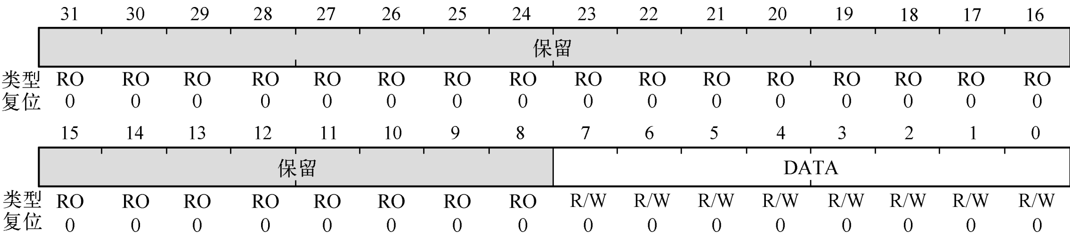


图 5-5　GPIO 数据寄存器（GPIODATA）

位/字段	名称	类型	复位	描述
31：8	保留	RO	0x0000.00	软件不可依赖保留位的值。为了兼容未来的产品，保留位的值在读取－修改－写入操作过程中应保持不变
7：0	DATA	R/W	0x00	GPIO 数据 该寄存器被虚拟地映射到地址空间的 256 个单元中。为便于通过单独的驱动器读写这些寄存器，从其读取的值和写入其中的值可通过 8 条地址线［9：2］屏蔽。读取该寄存器将返回其当前状态。写入该寄存器仅影响那些没有被 ADDR［9：2］屏蔽的位和被配置成输出的位

（3）GPIO 方向寄存器（GPIODIR），偏移量 0x400

在 GPIODIR 寄存器中的某位被置位会将相应的引脚配置成输出，而清零的话就是配置为输入。复位时，所有的位都被清零，这意味着所有 GPIO 端口默认为输入状态，如图 5-6 所示。

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	保留															
类型	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
复位	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	保留								DIR							
类型	RO	RO	RO	RO	RO	RO	RO	RO	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
复位	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 5-6　GPIO 方向寄存器（GPIODIR）

位/字段	名称	类型	复位	描述
31：8	保留	RO	0x0000.00	软件不可依赖保留位的值。为了兼容未来的产品，保留位的值在读取－修改－写入操作过程中应保持不变
7：0	DIR	R/W	0x00	GPIO 数据方向 值　描述 0　相应的引脚为输入 1　相应的引脚为输出

（4）GPIO 中断事件寄存器（GPIOIEV），偏移量 0x40C

当 GPIOIEV 寄存器中某位被置位时，相应的引脚由上升沿或是高电平触发中断，具体由 GPIO 中断检测寄存器（GPIOIS）中的位控制。清零一个位则表示相应的引脚由下降沿或是低电平触发，具体还是由 GPIOIS 中的位控制。复位时所有的位都是清零的，如图 5-7 所示。

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	保留															
类型	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
复位	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	保留								IEV							
类型	RO	RO	RO	RO	RO	RO	RO	RO	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
复位	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 5-7　GPIO 中断事件寄存器（GPIOIEV）

位/字段	名称	类型	复位	描述
31：8	保留	RO	0x0000.00	软件不可依赖保留位的值。为了兼容未来的产品，保留位的值在读取－修改－写入操作过程中应保持不变
7：0	IEV	R/W	0x00	GPIO 中断事件 值　描述 0　相应引脚上的下降沿或低电平触发中断 1　相应引脚上的上升沿或高电平触发中断

（5）GPIO 复用功能选择寄存器（GPIOAFSEL），偏移量 0x420

如果某位被清零，则表示相应的引脚用作 GPIO 功能，并受 GPIO 寄存器控制。某位被置位则表示相应的 GPIO 线受相关外设控制。每个 GPIO 上都有几个不同的复用外设功能。可以通过 GPIO 端口控制寄存器（GPIOPCTL）来选择其中的一个所需的功能，如图 5-8 所示。

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	保留															
类型	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
复位	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	保留								AFSEL							
类型	RO	RO	RO	RO	RO	RO	RO	RO	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
复位	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 5-8　复用功能选择寄存器（GPIOAFSEL）

位/字段	名称	类型	复位	描述
31：8	保留	RO	0x0000.00	软件不可依赖保留位的值。为了兼容未来的产品，保留位的值在读取－修改－写入操作过程中应保持不变
7：0	AFSEL	R/W	-	复用功能选择 值　描述 0　相应引脚为 GPIO 功能，受 GPIO 寄存器控制 1　相应引脚为外设功能，由复用硬件功能控制



5.2　GPIO 固件库函数

GPIO 的 API 分为三个函数组：

① 配置 GPIO 引脚。

② 中断处理。

③ 访问引脚状态。

（1）配置 GPIO 引脚的常用函数

- GPIODirModeSet( )。
- GPIOPadConfigSet( )。
- GPIOPinConfigure( )。
- GPIODirModeGet( )。
- GPIOPadConfigGet( )。
- GPIOPinTypeCAN( )。
- GPIOPinTypeComparator( )。
- GPIOPinTypeGPIOInput( )。
- GPIOPinTypeGPIOOutput( )。
- GPIOPinTypeGPIOOutputOD( )。

● GPIOPinTypeI2C( )。
● GPIOPinTypePWM( )。
● GPIOPinTypeQEI( )。
● GPIOPinTypeSSI( )。
● GPIOPinTypeTimer( )。
● GPIOPinTypeUART( )。

(2) 中断处理的常用函数

● GPIOIntTypeSet( )。
● GPIOIntTypeGet( )。
● GPIOIntEnable( )。
● GPIOIntDisable( )。
● GPIOIntStatus( )。
● GPIOIntClear( )。
● GPIOIntRegister( )。
● GPIOIntUnregister( )。

(3) 访问 GPIO 引脚状态的函数

● GPIOPinRead( )。
● GPIOPinWrite( )。

详细的 GPIO 固件库函数功能介绍请参考书后附录 C。



5.3 例程

本小节将以两个非常简单的闪烁灯实验来介绍基于寄存器和基于固件的 GPIO 模块的编程方法，以及较详细的调试过程。

1. 三只 LED 同时闪烁的程序（寄存器）

1) EK－TM4C123GXL 开发板三只 LED 的硬件连接图如图 5-9 所示。

2) LED 闪烁程序。LED 闪烁的程序流程图如图 5-10 所示。

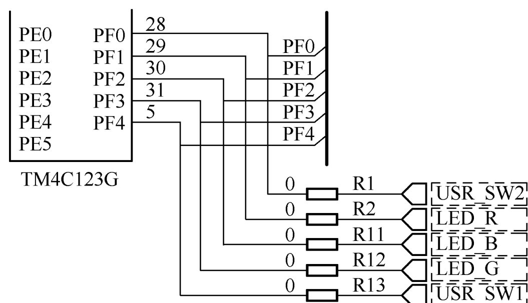


图 5-9　TM4C123GXL 开发板三只 LED 的硬件连接图

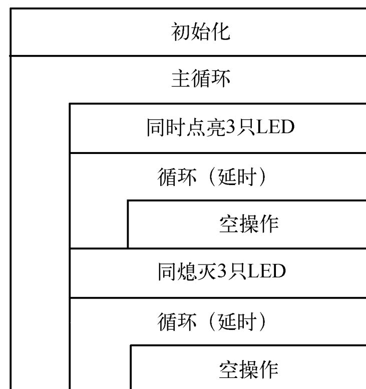


图 5-10　LED 闪烁程序流程图

```
// ***********************************************************
//文件名:threeblinky.c
//来源:根据 TI 例程改编
// ***********************************************************

#include <stdint.h>
#include "inc/tm4c123gh6pm.h"

int
main(void)
{
    volatile uint32_t ui32LEDloop;
    //
    //使能 GPIOF 端口
    //
    SYSCTL_RCGC2_R = SYSCTL_RCGC2_GPIOF;
    //
    //使能外设后插入几个周期的空读取操作
    //
    ui32LEDloop = SYSCTL_RCGC2_R;
    //
    //使能 PF1(LED_R)、PF2(LED_B)、PF3(LED_G)引脚
    //设置这些引脚为输出
    //使能这些引脚的数字功能
    //
    GPIO_PORTF_DIR_R = 0x0e;
    GPIO_PORTF_DEN_R = 0x0e;
    //
    //无限循环(主循环)
    //
    while(1)
    {
        //
        //点亮三只 LED 灯
        //
        GPIO_PORTF_DATA_R |= 0x0e;
        //
        //延时(循环)
        //
        for(ui32LEDloop = 0;ui32LEDloop < 200000;ui32LEDloop++)
        {
        }
        //
        //熄灭三只 LED 灯
        //
        GPIO_PORTF_DATA_R &= ~(0x0e);
        //
        //延时(循环 2)
```

```
            //
            for( ui32LEDloop = 0;ui32LEDloop < 200000;ui32LEDloop ++ )
            {
            }
        }
    }
```

3）在 Keil ARM 中创建 threeblinky_r 工程，如图 5-11 所示。

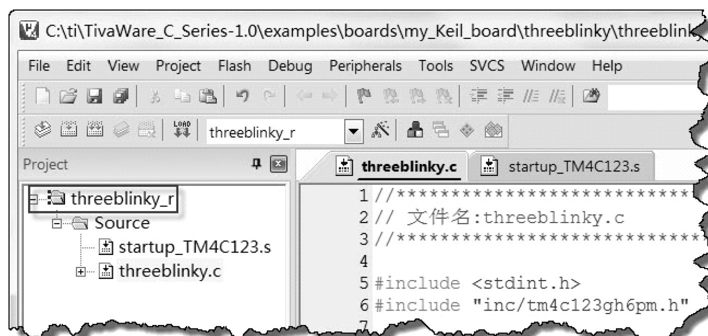


图 5-11　创建的闪烁灯工程（寄存器）

4）编译 threeblinky_r 工程生成可执行的 . axf 格式文件，如图 5-12 所示。

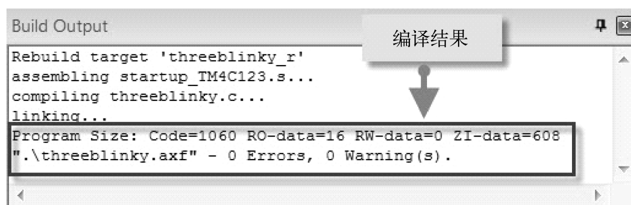


图 5-12　编译结果

5）在 EK－TM4C123GXL 板上对程序进行调试与测试。

① 调试。单击工具栏中的图标，调出调试窗口，如图 5-13 所示。

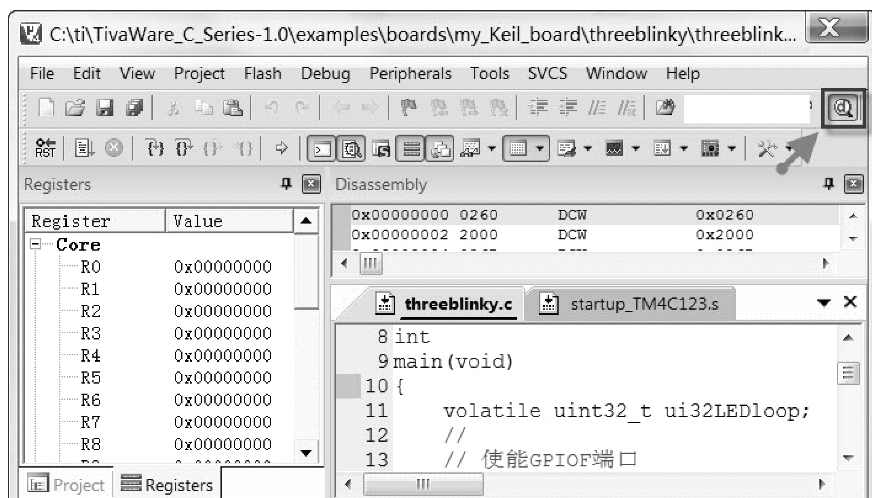


图 5-13　调试窗口

单击工具栏中的图标右侧的三角，按图 5-14 所示过程调出 GPIOF 端口，以便在调试中观察 DATA 的变化情况。

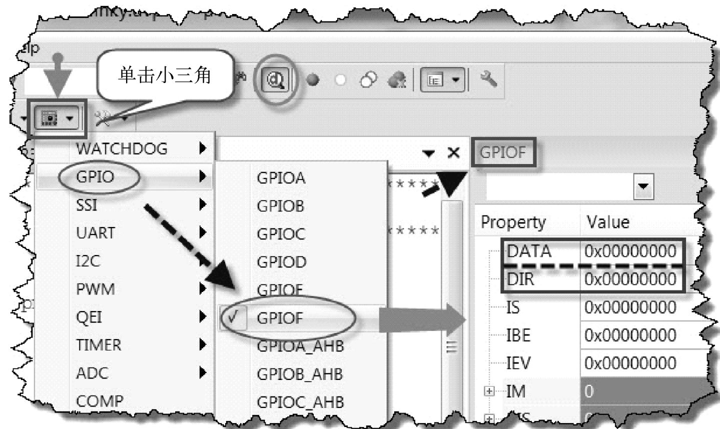


图 5-14　选中 GPIOF 端口的过程

右键单击将 ui32LEDloop 变量添加到观察窗口，如图 5-15 所示。

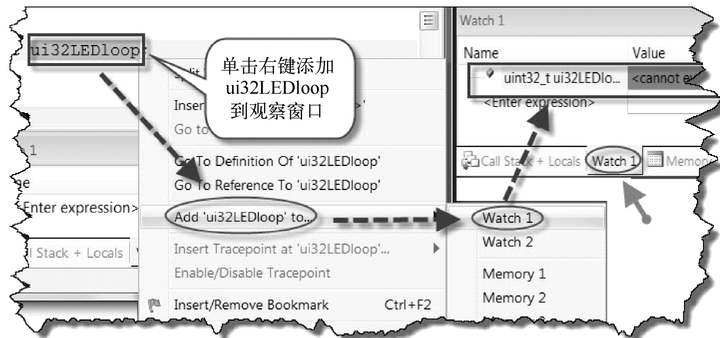


图 5-15　添加 ui32LEDloop 变量到观察窗口的过程

单击工具栏的全速运行按钮启动调试，其结果如图 5-16 所示。

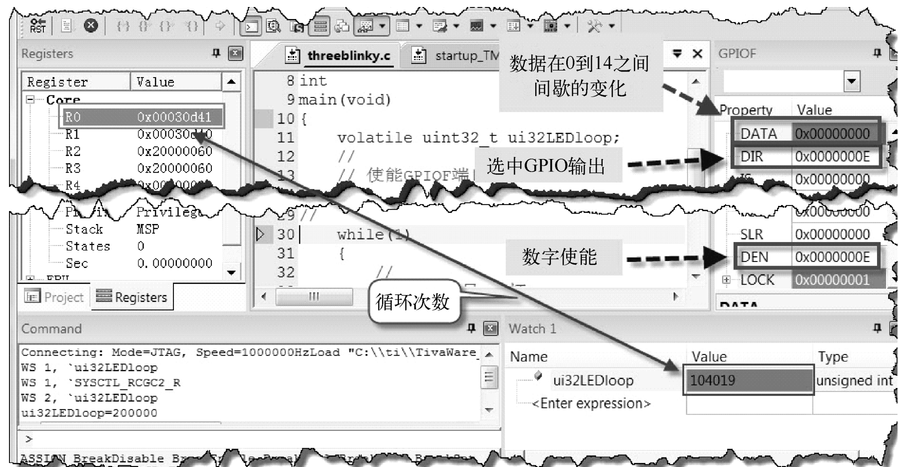


图 5-16　全速运行结果

从 DATA、DIR、RO 等寄存器中的值和 EK－TM4C123GXL 板上的运行结果来看，三灯同时闪烁程序是正确的。下面将以单步运行方式来验证程序的运行情况是否符合程序设计思想：

单击工具栏中的复位图标使循环计数寄存器 RO 清零，让程序回到 Startup_TM4C123G.s（如图 5-17 所示）的双三角位置。

```
threeblinky.c*    startup_TM4C123.s
256          ; in this project.
257          ;
258          MOVW    R0, #0xED88
259          MOVT    R0, #0xE000
260          LDR     R1, [R0]
261          ORR     R1, #0x00F00000
262          STR     R1, [R0]
...
268          ;
269          IMPORT  __main
270          B       __main
272  ;********************************
Text Editor   Configuration Wizard
```

图 5-17　初始化代码

然后单击〈F11〉键单步运行这段程序，当程序运行到 270 行（即 B_main）时，将光标放置在 threeblinky.c 的 main 函数处，然后按工具栏中的图标，让程序运行到光标处，使程序从 main 函数处开始执行，以便观察 DATA 和 ui32LEDloop 变量的变化情况（这时 DATA＝0x00000000、DIR＝0x00000000、R0＝0x20000060 和 ui32LEDloop＝0x00000000）。

按〈F11〉键继续单步执行，当程序运行到 ui32LEDloop＝SYSCTL_RCGC2_R 处时，R0＝0x00000020。

说明：此时已使能 GPIOF 端口。

然后继续单步执行，以观察程序运行的结果，当程序运行到 for(ui32LEDloop＝0;ui32LEDloop<200000;ui32LEDloop++)语句处时，为了使程序退出数万次的循环，可在命令行输入 ui32LEDloop＝200000 命令（如图 5-18 所示）使程序从空操作中退出。再继续单步执行以观察程序运行状况。

```
Command
ui32LEDloop=200000

>ui32LEDloop=200000
<C-style expression>  variable = <expression>
```

图 5-18　从命令行输入计数器最大值

小结：读者可从这个简单的例子中学会如何进行单步调试程序的方法。

② 测试。导入编译产生的 .axf 文件到 EK－TM4C123GXL 中，其运行及测试结果如图 5-19 所示。

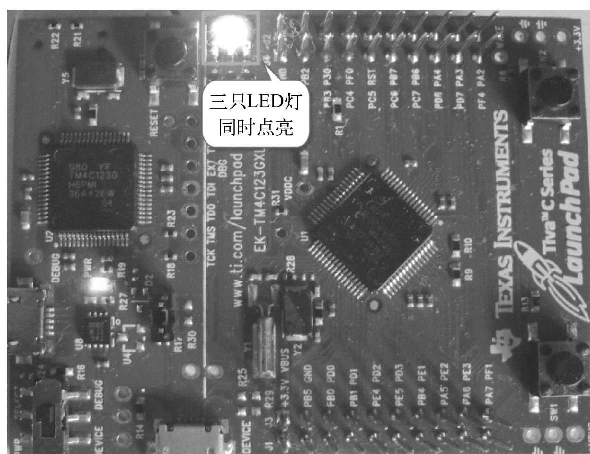


图 5-19　EK-TM4C123GXL 开发板测试结果

从图 5-19 的测试结果来看，开发板能同时发出比较炫目的白色光芒，验证了基于寄存器的三灯同时闪烁程序是正确的（R、G、B 三色发光管同时发光应该是白光，这里只是近似白色，这是由于三色的比较不对）。

2. 三只 LED 同时闪烁程序（固件库）

1）LED 程序。

```
//*************************************************************
//文件名:threeblinky_f.c
//来源:根据 TI 例程修改
//*************************************************************

#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"

uint8_t ui8PinLEDdata = 7;      //设置 LED 数据并初始化

int main(void)
{
    //
    //设置系统时钟:5 分配、使用 PLL、外接 16 MHz 晶振、使用主振荡器
    //
    SysCtlClockSet(SYSCTL_SYSDIV_5 | SYSCTL_USE_PLL | SYSCTL_XTAL_16MHZ |
                   SYSCTL_OSC_MAIN);
    //
    //使能外设 GPIOF 端口
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    //
```

```
        //将 FP1、FP2、FP3 端口设置为输出
        //
        GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3);
        //
        //无限循环
        //
        while(1)
        {
            //
            //同时点亮三只 LED
            //
             GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3,
                        ui8PinLEDdata);
            //
            //延时
            //
            SysCtlDelay(2000000);
            //
            //同时熄灭三只 LED
            //
            GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3,0x00);
            //
            //延时
            //
            SysCtlDelay(2000000);

        }
    }
```

2）创建 threeblinky_f 工程（固件库），如图 5-20 所示。

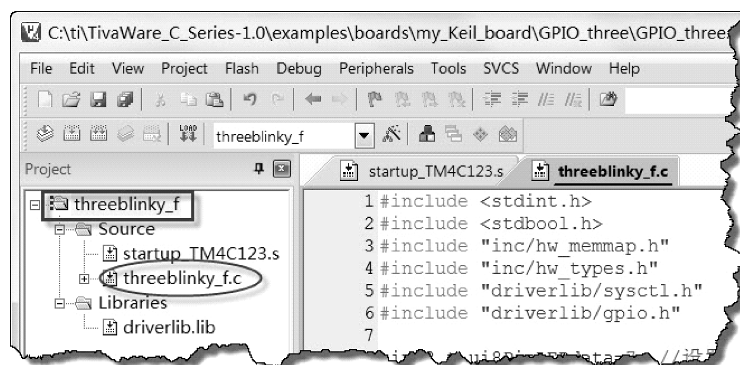


图 5-20　创建的 threeblinky_f 工程（固件库）

工程的创建过程如下：

① 添加 threeblinky. c 到工程中。

② 增加库函数组，并将驱动库添加到工程中。

③ 添加头文件及库文件搜索路径，如图 5-21 所示。

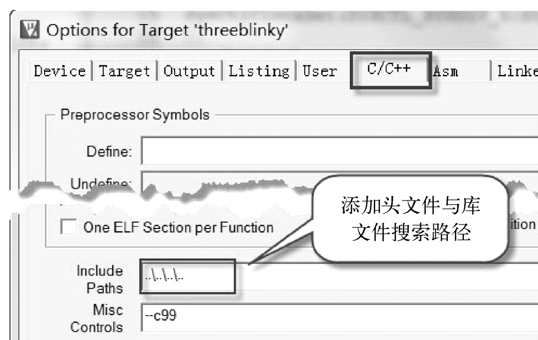


图 5-21　添加头文件及库文件搜索路径

说明：如果程序中有打✖的代码行存在，很多情况下就是由于程序无法找到头文件或库文件造成的，这时可以考虑重新设置搜索路径。

④ 添加 startup. s 文件。详细的操作过程请参考第 1 章 Keil ARM 的使用方法。

3）编译工程，其结果如图 5-22 所示。

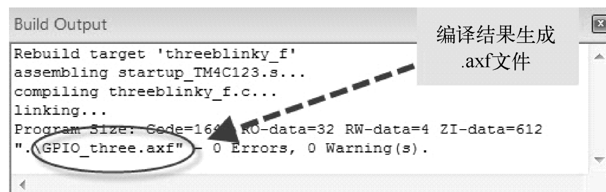


图 5-22　threeblinky_f 工程的编译结果

4）在 EK - TM4C123GXL 开发板上对程序进行测试。导入编译产生的 . axf 文件到 EK - TM4C123GXL 板中，其运行及测试结果如图 5-23 所示。

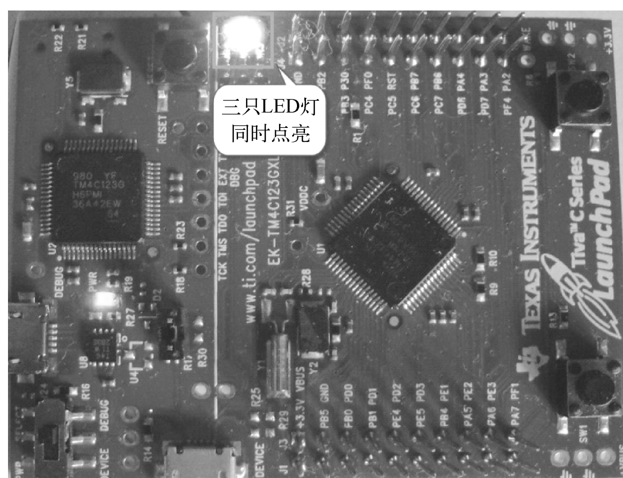


图 5-23　EK - TM4C123GXL 开发板测试结果

从图 5-23 的测试结果来看，开发板能同时发出比较炫目的白色光芒，验证了所创建的 threeblinky_f 工程实现了设计思想。

3. 在 Proteus 上测试流水灯程序

1）流水灯程序。

```
//**********************************************************
//liushuideng.c
//来源:根据 TI 例程改编
//功能:在 Proteus 虚拟硬件平台上实现对流水灯程序的测试
//作用:对无真实板卡和有某款开发板但缺少外设的读者验证自己
//所编程序之正误
//**********************************************************

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"
#include "driverlib/systick.h"

//**********************************************************
//
//设置用于 LED 显示的 GPIO 引脚
//
//**********************************************************
#define PINS          (GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3 | \
                       GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7)
//**********************************************************
//下面将使用固件库 GPIOPinTypeGPIOOutput()函数实现 8 LED 流水
//**********************************************************
int
main(void)
{
    //
    //设置系统时钟
    //
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_6MHZ);

    //
    //使能 GPIO 模块
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

    //
    //设置 GPIO 为输出端口
    //
    GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE,PINS );

while(1)
```

```
    {
        //
        //点亮 LED1
        //
        GPIOPinWrite(GPIO_PORTB_BASE,PINS,0x01);
        //
        //延迟,由于是软件模拟 LM3S301,因此运行速度比实际慢得多
        //这里延迟的时钟个数比真实 CPU 少得多,请读者注意
        //
        SysCtlDelay(15000);
        //
        //点亮 LED2
        //
        GPIOPinWrite(GPIO_PORTB_BASE,PINS,0x02);
        SysCtlDelay(15000);
        //
        //点亮 LED3
        //
        GPIOPinWrite(GPIO_PORTB_BASE,PINS,0x04);
        SysCtlDelay(15000);
        //
        //点亮 LED4
        //
        GPIOPinWrite(GPIO_PORTB_BASE,PINS,0x08);
        SysCtlDelay(15000);
        //
        //点亮 LED5
        //
        GPIOPinWrite(GPIO_PORTB_BASE,PINS,0x10);
        SysCtlDelay(15000);
        //
        //点亮 LED6
        //
        GPIOPinWrite(GPIO_PORTB_BASE,PINS,0x20);
        SysCtlDelay(15000);
        //
        //点亮 LED7
        //
        GPIOPinWrite(GPIO_PORTB_BASE,PINS,0x40);
        SysCtlDelay(15000);
        //
        //点亮 LED8
        //
        GPIOPinWrite(GPIO_PORTB_BASE,PINS,0x80);
        SysCtlDelay(15000);

    }
}
```

2）搭建测试电路图。流水灯程序的测试电路如图 5-24 所示。

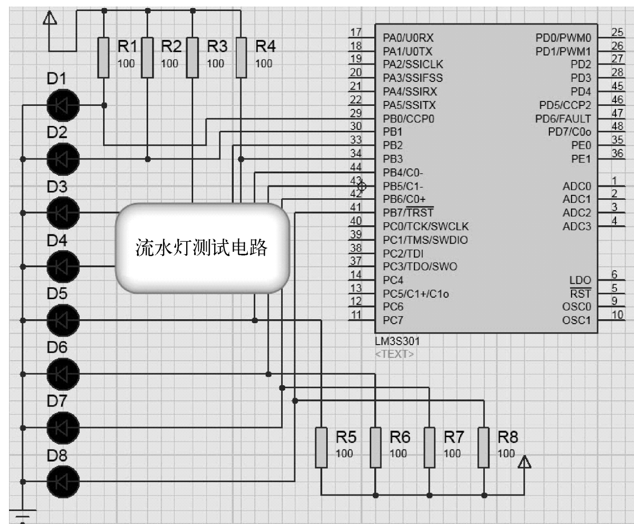


图 5-24　流水灯程序的测试电路

3）流水灯程序的测试结果。流水灯程序的测试结果如图 5-25 所示。

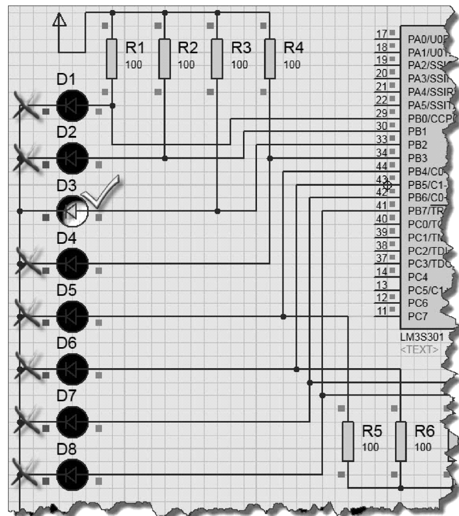


图 5-25　流水灯程序的测试结果

第 6 章

模拟比较器（COMP）

模拟比较器是一个比较两个模拟电压大小的外设，并提供一个逻辑输出信号作为比较的结果。比较器可以向器件引脚提供输出，以替代板上的模拟比较器。它也可以通过中断或触发向应用发出启动 ADC 转换的信号。中断产生逻辑和 ADC 触发是各自独立的。也就是说，即使中断在上升沿产生，ADC 也可在下降沿触发。

比较器的 API 提供了一组用于编程和使用模拟比较器的函数。此驱动程序包含在 driverlib/comp. c 中，driverlib/comp. h 包含 API 函数的定义。

本章的主要内容：

- COMP 单元
- COMP 固件库函数（请见书后附录 D）
- 例程



6.1 COMP 单元

6.1.1 COMP 特点

TM4C123GH6PM 微控制器提供两个独立集成的模拟比较器。它们具有如下特点：

1）可以比较外部输入引脚与外部输入引脚或内部可编程的参考电压。

2）比较器可执行测试电压与下列电压的比较：

① 单个外部独立的参考电压。

② 共享单个外部参考电压。

③ 共享内部参考电压。

6.1.2 COMP 模块框图

COMP 模块框图如图 6-1 所示。

6.1.3 信号描述

COMP 信号描述见表 6-1。

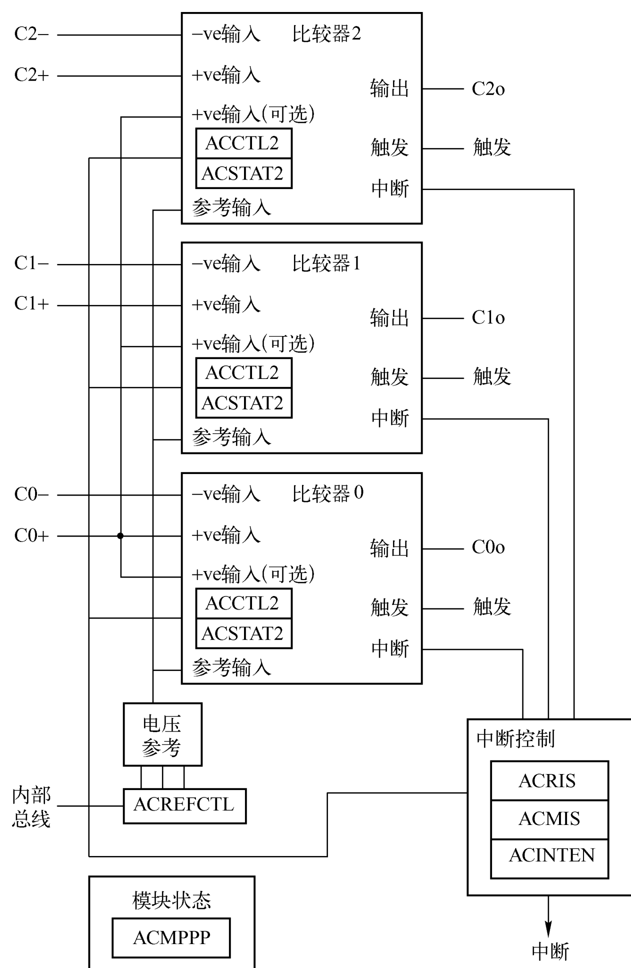


图 6-1　COMP 模块框图

注意： TM4C123GH6PM 微控制器只有两个模拟比较器。

表 6-1　COMP 信号（64LQFP）

引脚名称	引脚编号	引脚复用/引脚分配	引脚类型	缓冲区类型	描　　述
C0 +	14	PC6	I	模拟	模拟比较器 0 同相输入
C0 −	13	PC7	I	模拟	模拟比较器 0 反相输入
C0o	28	PF0(9)	O	TTL	模拟比较器 0 输出
C1 +	15	PC5	I	模拟	模拟比较器 1 同相输入
C1 −	16	PC4	I	模拟	模拟比较器 1 反相输入
C1o	29	PF1(9)	O	TTL	模拟比较器 1 输出

6.1.4　功能简介

比较器比较 VIN − 和 VIN + 的输入，然后输出 VOUT。

VIN − < VIN + , VOUT = 1

VIN－＞VIN＋，VOUT＝0

VIN－的信号源是一个外部输入（Cn－），其中 n 为模拟比较器的编号。除了外部的输入源（Cn＋）之外，VIN＋的输入源还可为 C0＋，或为一个内部参考电压（V_{IREF}），如图 6-2 所示。

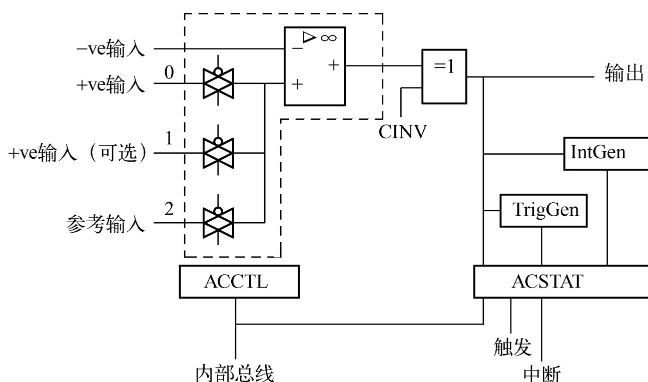


图 6-2　比较单元的结构

比较器是通过两个状态/控制寄存器（即模拟比较器控制寄存器（ACCTL）和模拟比较器状态寄存器（ACSTAT））来配置的。而内部参考电压则是通过一个控制寄存器（即模拟比较器参考电压控制寄存器（ACREFCTL））来配置的。中断的状态和控制则需通过三个寄存器（即模拟比较器中断状态寄存器（ACMIS）、模拟比较原始中断状态寄存器（ACRIS）和模拟比较器中断使能寄存器（ACINTEN））来配置。

一般情况下，比较器的输出通过 ACCTL 寄存器中的 ISEN 位在内部产生一个中断，此输出也可以用于驱动外部引脚（Cno）或产生模数转换器（ADC）触发信号。

注意：在使用模拟比较器之前，ACCTL 寄存器中的 ASRCP 位必须置位。

内部参考电压由单一配置寄存器 ACREFCTL 控制，其结构如图 6-3 所示。

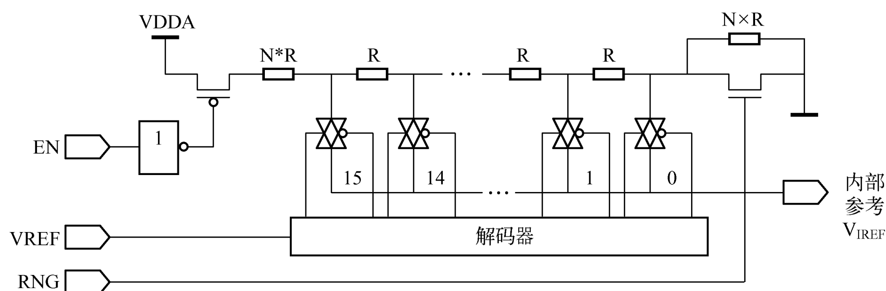


图 6-3　比较器内部参考电压结构

可根据 ACREFCTL 寄存器中的 RNG 位，采用两种模式（低电平或高电平）来编程内部参考电压。当 RNG 位清零时，内部参考电压采用高电平模式；当 RNG 位置位时，则内部参考电压采用低电平模式。

在每种模式下，内部参考电压（V_{IREF}）有 16 级预编程阈值或步长。用于与外部输入电

压比较的阈值电压，可使用 ACREFCTL 寄存器中的 VREF 字段来选择。内部参考电压与 ACREFCTL 字段值见表 6-2。

表 6-2　内部参考电压与 ACREFCTL 字段值

ACREFCTL 寄存器		基于 VREF 字段值的输出参考电压
EN 位值	RNG 位值	
EN＝0	RNG＝X	VREF 的值为 0 V(GND)；建议使用 RNG＝1 和 VREF＝0 来获得低噪声的接地参考
EN＝1	RNG＝0	V_{IREF}高电平：16 级阈值电压 VREF＝0、1、2 、…、15 理想的起始电压（VREF＝0）：VDDA /4.2 理想的步长：VDDA/ 29.4 理想的 V_{IREF}阈值：V_{IREF}（VREF）＝VDDA / 4.2＋VREF ＊（VDDA/ 29.4，其中 VREF＝0、1、2 、…、15
	RNG＝1	V_{IREF}低电平：16 级阈值电压 VREF＝0、1、2 、…、15 理想的起始电压（VREF＝0）：0 V 理想的步长：VDDA/ 22.12 理想的 V_{IREF} 阈值：V_{IREF}（VREF）＝VREF ＊（VDDA/ 22.12)，其中 VREF＝0、1、2 、…、15

6.1.5　寄存器映射

表 6-3 中列出了模拟比较器的寄存器映射，表中所列偏移量是寄存器地址相对于模拟比较器基址（0x4003.C000）的增量（十六进制地址）。注意，在对这些寄存器编程之前必须先行使能模拟比较器的时钟，且在该时钟使能后，必须等待至少 3 个系统时钟才可访问模拟比较器寄存器。

表 6-3　模拟比较器寄存器映射

偏移量	名称	类型	复位	描述
0x000	ACMIS	R/W1C	0x0000.0000	模拟比较器屏蔽中断状态寄存器
0x004	ACRIS	RO	0x0000.0000	模拟比较器原始中断状态寄存器
0x008	ACINTEN	R/W	0x0000.0000	模拟比较器中断启用寄存器
0x010	ACREFCTL	R/W	0x0000.0000	模拟比较器参考电压控制寄存器
0x020	ACSTAT0	RO	0x0000.0000	模拟比较器状态寄存器 0
0x024	ACCTL0	R/W	0x0000.0000	模拟比较器控制寄存器 0
0x040	ACSTAT1	RO	0x0000.0000	模拟比较器状态寄存器 1
0x044	ACCTL1	R/W	0x0000.0000	模拟比较器控制寄存器 1
0xFC0	ACMPPP	RO	0x0003.0003	模拟比较器外设属性寄存器



6.2　COMP 固件库函数

比较器的 API 就像比较器本身一样简单，主要由两组函数组成：

① 配置比较器和读取输出函数。

② 比较器中断处理函数。

(1) 配置比较器和读取输出函数

- ComparatorConfigure( )。
- ComparatorRefSet( )。
- ComparatorValueGet( )。

(2) 比较器中断处理函数

- ComparatorIntRegister( )。
- ComparatorIntUnregister( )。
- ComparatorIntEnable( )。
- ComparatorIntDisable( )。
- ComparatorIntStatus( )。
- ComparatorIntClear( )。

详细的模拟比较器固件库函数功能说明请参考书后附录 D。



6.3 例程

本小节两个示例介绍模拟比较器的编程与调试方法。

1. 同相输入端的参考电压为内部参考电压（在 Proteus 中测试）

(1) 比较器的硬件接线图

硬件接线图如图 6-4 所示。

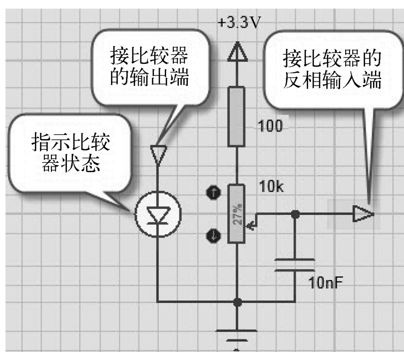


图 6-4 程序硬件接线图

(2) 编写比较器程序

```
// ***************************************************************************
//!程序名:comparator. c
//!来源: 根据 TI 例程改编
//!程序功能说明:模拟比较器的操作方法
//!即在比较器 0 的反相输入端接一个电位器作为外部输入电压源与内部产生的 1. 7875V
//!参考电压进行比较,并根据其输出改变中断来翻转端口 B0 上 LED 的状态。当检测到
//!比较器输出的上升沿时,中断服务程序点亮 LED,而检测到比较器的下降沿时中断服务
//!程序熄灭 LED
//! 比较器原理:          VIN - < VIN + ,VOUT = 1
```

```
//!                         VIN - > VIN + , VOUT = 0
//! 调试思想:
//!                     上升沿:VIN - > VIN + → VIN - < VIN + ,即 0 → 1 ↑
//!                     下降沿:VIN - < VIN + → VIN - > VIN + ,即 1 → 0 ↓
// ****************************************************************************

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/comp.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"

// ****************************************************************************
//如果驱动程序库发生错误,将调用该错误例程
// ****************************************************************************
#ifdef DEBUG
void
__error__(char *pcFilename,unsigned long ulLine)
{
}
#endif

// ****************************************************************************
//比较器中断处理程序
// ****************************************************************************
void
CompIntHandler(void)
{
    //
    //清除比较器中断
    //
    ComparatorIntClear(COMP_BASE,0);

    //
    //根据比较器当前的输出值来设置 GPIO B0 口的值
    //
    if(ComparatorValueGet(COMP_BASE,0) == true)
    {
        GPIOPinWrite(GPIO_PORTB_BASE,GPIO_PIN_0,GPIO_PIN_0);
    }
    else
    {
        GPIOPinWrite(GPIO_PORTB_BASE,GPIO_PIN_0,0);
    }
}
```

```
//*****************************************************************************
//这个例子演示了如何设置一个模拟比较器以及触发输出变化中断
//*****************************************************************************
int
main(void)
{
    //
    //设置直接从晶振运行的时钟
    //
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_6MHZ);

    //
    //使能要用到的外设
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_COMP0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    //
    //使能处理器中断
    //
    IntMasterEnable();
    //
    //设置 GPIO B0 为输出,在此端口外接一只 LED 以指示当前比较器的输出值
    //
    GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE,GPIO_PIN_0);

    //
    //配置反相输出端(C0 - )
    //
    GPIOPinTypeComparator(GPIO_PORTB_BASE,GPIO_PIN_4);

    //
    //设置内部参考电压为 1.7875 V
    //
    ComparatorRefSet(COMP_BASE,COMP_REF_1_7875V);

    //
    //配置比较器使用内部参考电压和在输出的上升沿和下降沿产生中断
    //
    ComparatorConfigure(COMP_BASE,0,COMP_INT_BOTH | COMP_ASRCP_REF);

    //
    //使能比较器中断
    //
    IntEnable(INT_COMP0);
    ComparatorIntEnable(COMP_BASE,0);

    //
```

```
        //无限循环用于 LED 跟踪比较器的输出状态
        //
        while(1)
        {
        }
    }
```

（3）在 CCS6 中创建 comparator 工程

创建的 comparator 工程如图 6-5 所示。

详细的操作过程请参考第 1 章有关 CCS 6 的使用方法部分。

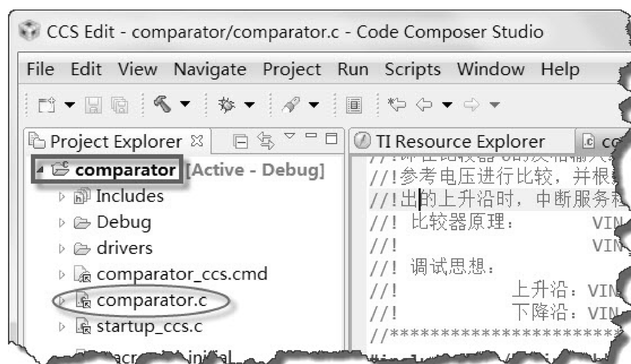


图 6-5　创建的 comparator 工程

（4）编译 comparator 工程

① 指定头文件和库文件的搜索路径，如图 6-6 所示。

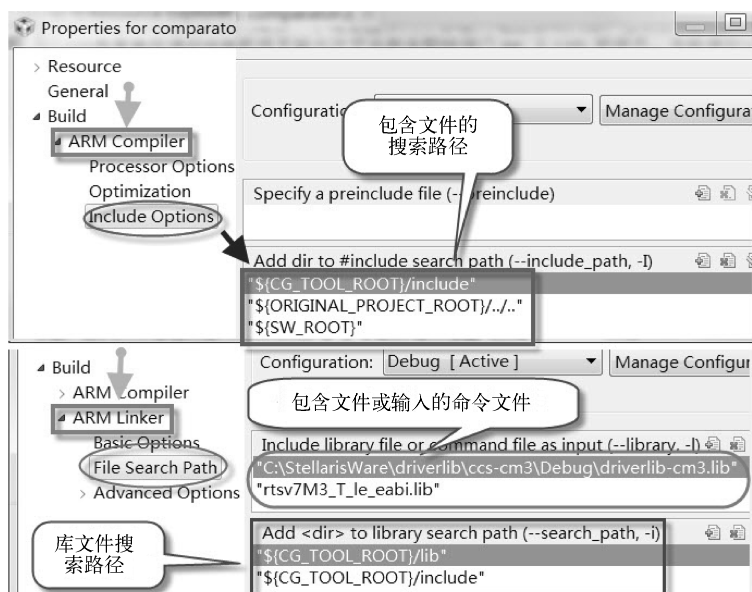


图 6-6　头文件和库文件的搜索路径

② 将 CCS6 编译产生的可执行文件改为 .elf 格式，如图 6-7 所示。

③ 编译 comparator 工程，生成 .elf 格式的可执行文件，如图 6-8 所示。

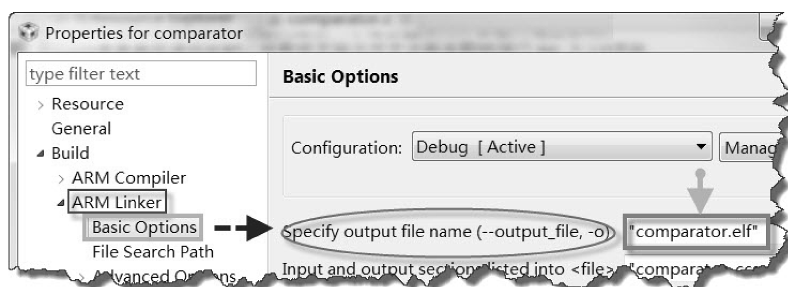


图 6-7　将编译生成的可执行文件改为 . elf 格式

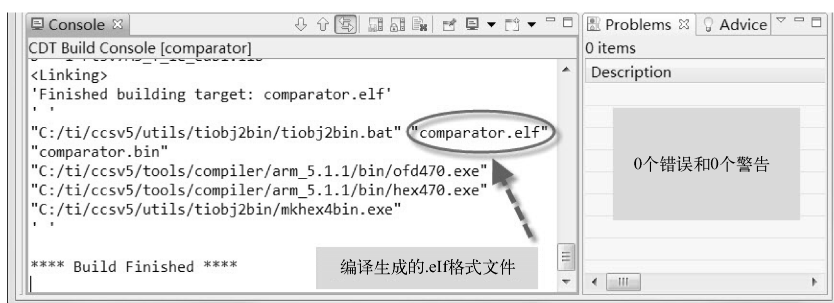


图 6-8　工程编译结果生成 . elf 格式文件

（5）在 Proteus 8. 1 中对工程的编译结果进行测试

① 搭建虚拟测试电路，如图 6-9 所示。

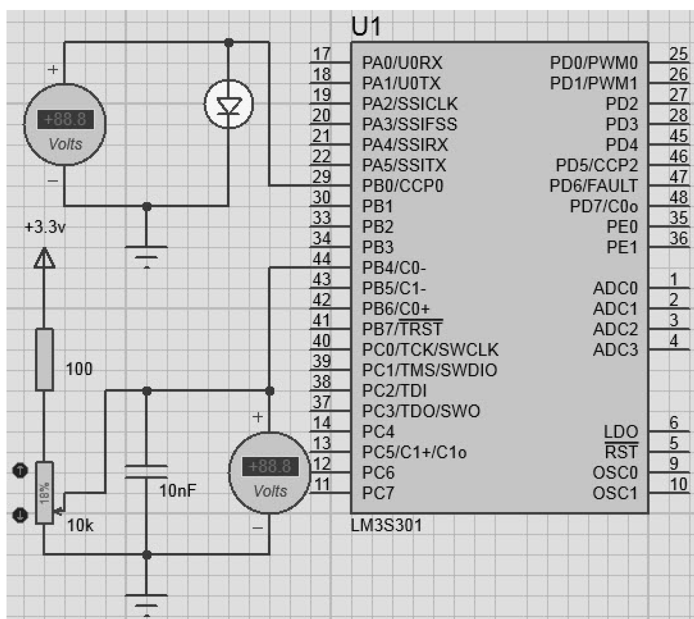


图 6-9　比较器程序的虚拟测试电路

② 导入编译产生的 . elf 文件到测试电路中，再将电位器中心触点移到最上面，然后启动 Proteus 测试，这时比较器的输出为 0，如图 6-10 所示。

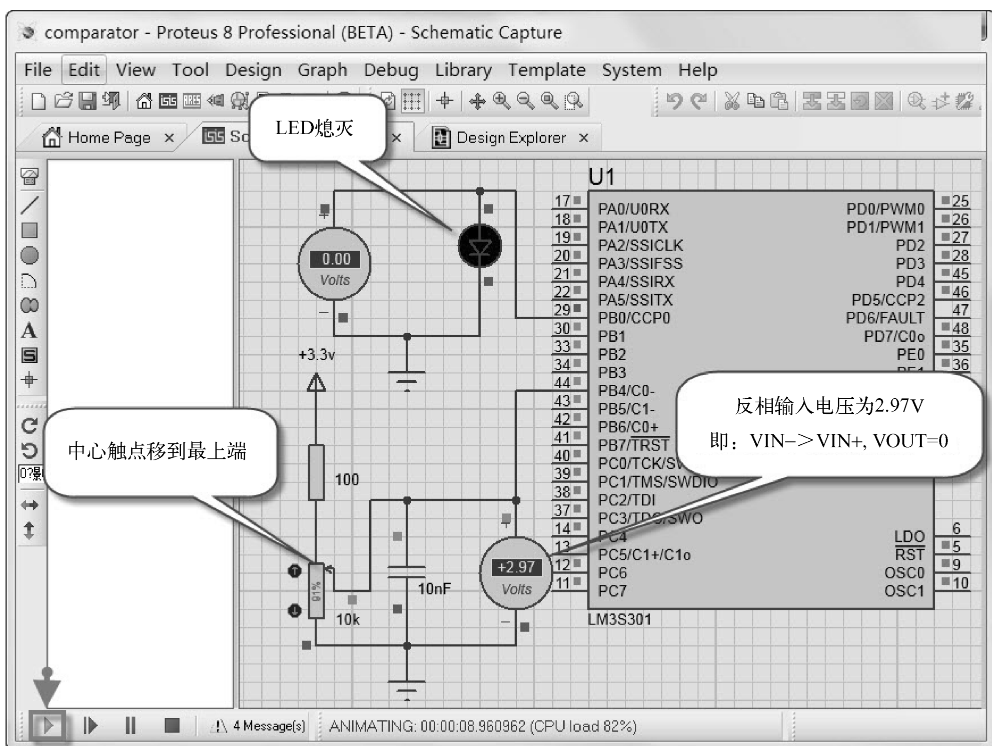


图 6-10　模拟比较器的输出为 0

为了能模拟一个上升沿，应该让比较器的输出为 1，即 VIN − < VIN + ,VOUT = 1，此时需将电位器的中心触点往下移，使 VIN − < VIN + 。一旦比较器输出高电平，将形成一个上升沿（即 0→1），从而触发中断点亮 LED，如图 6-11 所示。

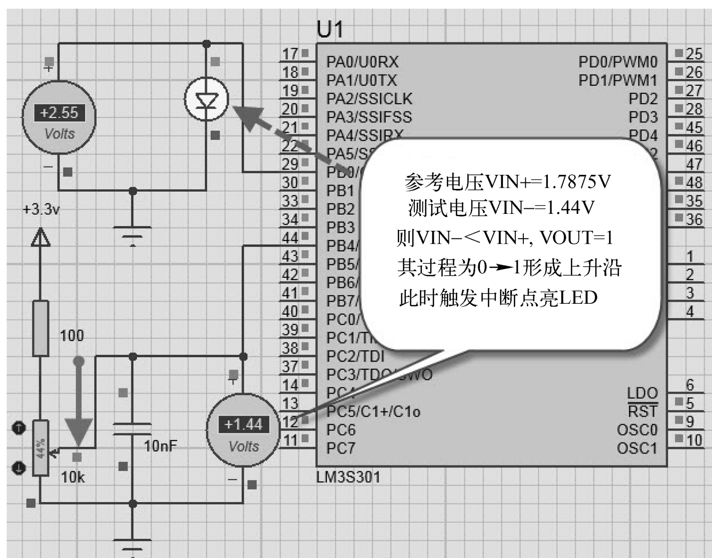


图 6-11　模拟上升沿触发中断点亮 LED

接着再把电位器中心触点往上移使 VIN - > VIN +，即模拟比较器输出 0（即 1→0），形成下降沿，其测试结果如图 6-12 所示。

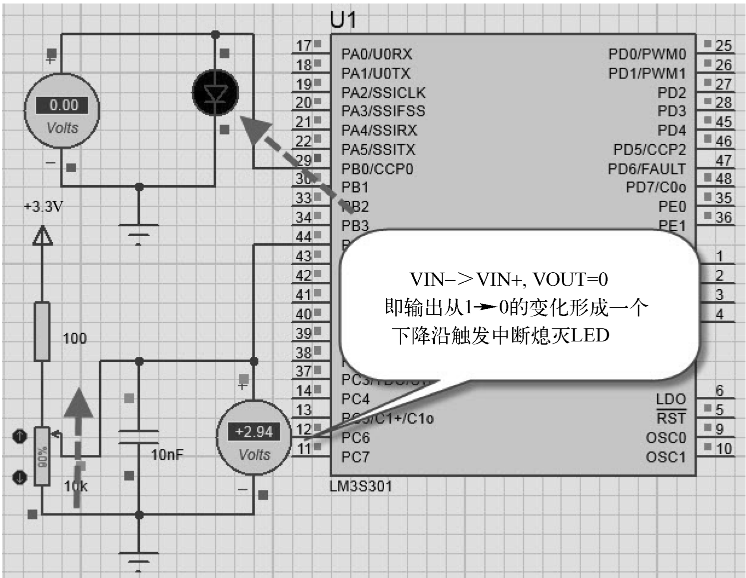


图 6-12　模拟下降沿熄灭 LED 灯

从以上的测试结果来看，比较器程序实现了所要求的功能，程序设计正确。

2. 在 EK - TM4C123GXL 板上对程序进行调试

这部分仅给出程序，测试实验请读者按照上例自行完成。

（1）比较器例程连线图

比较器例程连线图如图 6-13 所示。

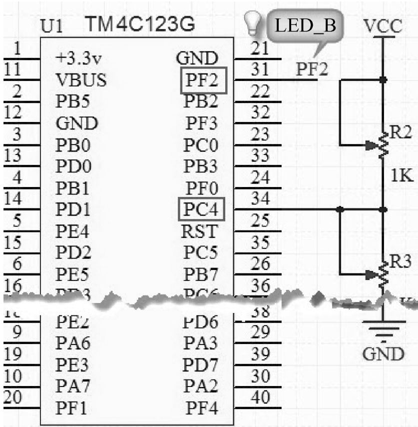


图 6-13　比较器例程连线图

（2）编写 comp. c 程序

```
//***************************************************************************
//文件名：comp. c
//来源：实验室改编
```

```
//!功能描述:
//!由比较器 1 负输入端输入模拟电压信号(GPIOC4),该信号与内部基准电压 1.753125V 相比
//!较。如果比较器 1 输出端产生上升沿或下降沿,就触发中断处理程序。中断处理程序中,如
//!果判断比较器 1 输出端为 1,则点亮接在 GPIOF2 的 LED;如果比较器 1 输出端为 0,则熄灭
//!GPIOF2 的 LED
//!硬件接口:
//!EK-TM4C123GXL,PC4 接电位器产生输入电压。一旦电压低于 1.753125V,则 LED 蓝灯亮
//***************************************************************************

#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/tm4c123gh6pm.h"
#include "driverlib/comp.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"

//***************************************************************************
//声明比较器 1 中断处理程序
//***************************************************************************
void CompIntHandler(void);
//***************************************************************************
//如果驱动程序库发生错误,将调用该错误例程
//***************************************************************************
#ifdef DEBUG
void
__error__(char *pcFilename,unsigned long ulLine)
{
}
#endif
//***************************************************************************
//比较器 1 中断处理程序
//***************************************************************************
void
CompIntHandler(void)
{
    bool Comp_out;
    //
    //清除比较器 1 中断标志
    //
    ComparatorIntClear(COMP_BASE,1);
    //
    //根据比较器 1 输出值来设置 GPIOF2 的值
    //tm4c123gxl 板子上,GPIOF2 接 LED_BLUE,GPIOF2 位高电平时,LED 蓝灯亮
    //
    Comp_out = ComparatorValueGet(COMP_BASE,1);
    if(Comp_out == 1)
```

```
    {
        GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2,0x04);
    }
    else
    {
        GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2,0x00);
    }
}

int main(void)
{
    SysCtlClockSet(SYSCTL_SYSDIV_10 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_
XTAL_16MHZ);
    //
    //使能比较器模块
    //注意:无论是比较器 0 还是比较器 1,使能函数中的参数都必须是 SYSCTL_PERIPH_COMP0
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_COMP0);
    //
    //配置比较器 1
    //比较器不触发 ADC;比较器输出端的上升沿和下降沿都将产生中断
    //比较器 1 正输入端采用内部基准电压;比较器 1 输出端信号正常输出(不翻转)
    //
    ComparatorConfigure(COMP_BASE,1,COMP_TRIG_NONE | COMP_INT_BOTH | COMP_ASRCP_
REF | COMP_OUTPUT_NORMAL);
    //
    //设置比较器内部基准电压值为 1.753125V
    //
    ComparatorRefSet(COMP_BASE,COMP_REF_1_753125V);
    //
    //使能 GPIOF 和 GPIOC 模块,GPIOF2 作为 LED 灯的输出;GPIOC4 为比较器 1 的负输入端
    //注意:如果片上外设用到 GPIO 端口,必须使能相应的 GPIO 模块
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
    //
    //将 GPIOF2 设置为输出模式
    //
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,GPIO_PIN_2);
    //
    //将 GPIOC4 设置为比较器 1 的负输入端
    //
    GPIOPinTypeComparator(GPIO_PORTC_BASE,GPIO_PIN_4);
    //
    //使能比较器 1 的中断
    //
    ComparatorIntEnable(COMP_BASE,1);
    //
```

```
    //注册比较器 1 的中断处理程序
    //
    ComparatorIntRegister(COMP_BASE,1,CompIntHandler);
    //
    //使能全局中断
    //
    IntMasterEnable();

    //
    //进入无限循环
    //
    while(1);
}
```

（3）建立工程及调试步骤

将上述程序下载到 EK－TM4C123GXL 板中，按运行按钮，然后调节与 PC4 引脚所连接的电位器，使输入电压信号发生改变，可观察 LaunchPad 板上蓝色 LED 灯的亮灭，如图 6-14a、b 所示。

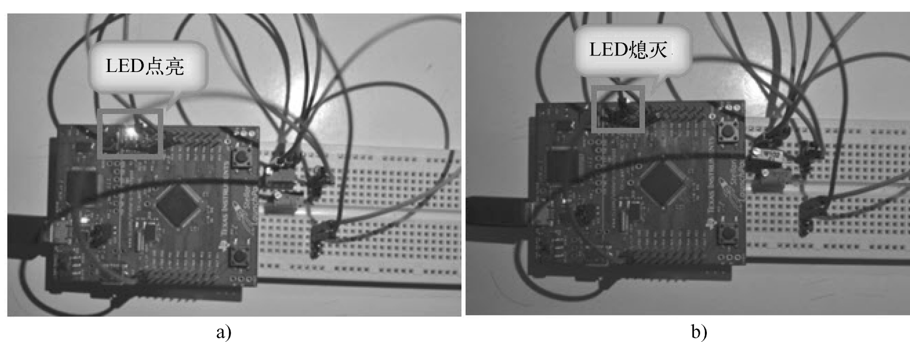


图 6-14　蓝色 LED 灯的亮灭

a）LED 蓝灯亮　b）LED 蓝灯灭

第7章

系统定时与中断控制

学习本章内容前，先介绍一下系统定时器和中断控制器。

（1）系统定时器（SysTick）

TM4C123GH6PM 控制器内核集成了一个系统定时器（SysTick），它是一个简单的定时器，为 Cortex－M 微处理器 NVIC 控制器中的一部分，其作用是为 RTOS 提供一个周期性中断。另外它也可用于其他简单的定时操作，提供一个简单的、控制灵活的 24 位递减计数器，还具有写入即清零、过零自动重载等特性。在调用 SysTick 中断处理程序时，会通过 NVIC 自动清除 SysTick 的中断源。

1）系统定时器的应用如下：

① 作为 RTOS 的节拍定时器，可按编程频率定时触发，调用系统定时器处理子程序。

② 可作为使用系统时钟的高速报警定时器。

③ 速率可变的报警或信号定时器。

④ 可作为简易计数器，用于测量任务的完成时刻、总体耗时等。

⑤ 可实现依据失配/匹配周期的内部时钟源控制。

2）系统定时器所包含的 3 个寄存器如下：

① 控制和状态（STCTRL）：控制和状态计数器用于配置时钟、使能计数器、使能中断与确定计数器的状态。

② 重载值（STRELOAD）：计数器的重载，用于计数器的重装值（Wrap Value）。

③ 当前值（STCURRENT）：计数器的当前值。

3）工作过程描述如下：

在使能系统定时器后，计数器对于每个时钟将递减一次，从重载值依次递减到 0，然后在下一个时钟沿翻转，再对每个时钟递减一次，周而复始。如果将 STRELOAD 寄存器清零，将在下次重载时禁止该计数器。当计数器递减到 0 时，COUNT 标志位将被置位。读取 COUNT 标志位后将使其自动清零。

若写 STCURRENT 寄存器，可将其清零，并且还可清除 COUNT 标志位。写该寄存器时不会触发 SysTick 异常逻辑，而读取该寄存器时，其返回值是此寄存器被访问时刻的内容。

SysTick 计数器按照系统时钟或精确内部振荡器（PIOSC）4 分频运行。若在某些低功耗模式下停止供给该时钟信号，将会使系统定时器上的计数器也会停止运行。在 SysTick 控制和状态寄存器（STCTRL）中置位 CLK_SRC 位，并保证深度睡眠时钟配置寄存器（DSLP-CLKCFG）中的 PIOSCPD 位被清零，以便 SysTick 在深度休眠模式中保持运行状态。在软件访问 SysTick 的寄存器时，应保证始终采用字对齐操作访问。

4）复位后的操作。在复位时，由于 SysTick 计数器的重载值和当前值未被定义，则 Sy-

sTick 计数器的正确初始化过程如下：

① 编程 STRELOAD 寄存器中的值。

② 写入任意值到 STCURRENT 寄存器中使其清零。

③ 将 STCTRL 寄存器配置成所要求的操作。

注意：在调试时，若处理器暂停，该计数器将不会递减。

此驱动程序包含在 driverlib/systick. c 中，driverlib/systick. h 包含应用程序使用的 API 的定义。

（2）嵌套式向量化中断控制器（NVIC）

中断为 CPU 实时地处理内部或外部事件的一种机制。当发生某种事件时，CPU 将暂停当前的程序执行，转而去处理中断事件。当中断服务程序结束后，又会返回到前面程序的中断处，重新开始往下执行。

TM4C123GH6PM 控制器包含 ARM 嵌套向量中断控制器（Nested Vectored Interrupt Controller，NVIC）。NVIC 和 TM4C 处理器在处理模式中可对所有异常进行优先级划分和处理。且在处理异常时处理器状态将会自动保存到堆栈中，而当中断服务程序（ISR）结束时又会自动恢复。中断向量的读取与状态保存同步，可高效地进入中断。处理器支持尾链技术，背对背的中断执行可免去入栈和出栈的时间开销。并可软件设置 7 个异常（系统处理器）和 78 个中断的 8 级优先级。

中断处理程序的配置方式：在编译时静态配置；在运行时动态配置。

静态配置可通过编辑使能代码中的中断程序表来实现。在静态配置时，必须首先通过使能 IntEnable（）函数才能开启 NVIC 中断，处理器才可响应该中断。

对于中断在运行时的动态配置，可使用 IntRegister( ) 或相关驱动函数进行。当使用 IntRegister( ) 时，首先必须使能中断；当使用单个相关函数时，IntEnable( ) 由驱动函数调用，而无需由程序来调用。

中断控制器 API 提供了一组处理 NVIC 的函数。该函数用于使能和禁止中断、注册中断处理程序和设置中断优先级。

此驱动程序包含在 driverlib/interrupt. c 中，driverlib/interrupt. h 包含应用程序使用的 API 的定义。

本章主要内容：

- 中断控制器
- 系统定时器
- 中断及系统定时器固件库函数（见附录 E）
- 系统定时器中断实例



7. 1　NVIC 模块

7. 1. 1　NVIC 模块的特点

NVIC 模块支持的一些主要特点如下：

① 78 个中断。

② 每个中断分为 0 ~ 7 个优先级均可编程，0 为最高优先级。
③ 低延时异常和中断处理。
④ 中断信号的电平检测和脉冲检测。
⑤ 动态重新分配中断的优先级。
⑥ 分组的优先级，被划分为组优先级字段和子优先级字段。
⑦ 支持背靠背的中断尾链技术。
⑧ 提供一个外部不可屏蔽中断（NMI）。

7.1.2 功能描述

1. 电平式与脉冲式中断

处理器支持电平式中断及脉冲式中断。脉冲式中断通常又称为边沿触发中断。

对于电平式中断而言，只要外设产生中断信号就会始终保持处于触发状态，直到外设中断信号复原后才不再触发。一般来说，这需要 ISR（中断服务子程序）对外设进行操作，使得外设不再产生中断请求信号。脉冲中断是在处理器时钟的上升沿同步采样的中断信号。为了确保 NVIC 能够检测到中断，外设所产生的中断信号必须保持至少一个时钟周期，在此期间 NVIC 可检测到脉冲并锁存中断。

处理器进入 ISR 后，将自动清除该中断的挂起状态。对于电平式中断，如果处理器从 ISR 返回后，中断信号仍未复原，则中断将再次挂起，于是处理器必须再次运行其 ISR。因此，外设可以像这样保持中断信号持续享用服务，直到其不再需要服务为止。

2. 中断的硬件控制及软件控制

1）Cortex – M4 锁存所有中断。当满足以下其中一个条件时，外设中断将被挂起：

① NVIC 检测到中断信号为高电平，且该中断未处于激活状态。
② NVIC 检测到中断信号的上升沿。
③ 软件写入相应的中断设置挂起寄存器的位，或者写入软件触发中断寄存器（SWTRIG）使软件产生的中断挂起。

2）中断挂起后到出现以下条件之一时，将一直保持挂起状态：

① 处理器进入该中断的 ISR，将中断状态由挂起变更为激活。

- 对于电平中断，当处理器从 ISR 返回后，NVIC 将重新采样中断信号。若检测到的中断信号有效，则中断状态将再次处于挂起状态，导致处理器立即重新回到 ISR；否则，中断状态将变为未激活状态。
- 对于脉冲中断，处理器将连续监控中断信号。如果检测到中断脉冲信号，就会将中断状态变更为挂起和激活。因此，当处理器从 ISR 返回后，中断状态将再次被挂起，导致处理器立即重新进入 ISR。

若处理器在 ISR 中并未产生中断脉冲信号，则当处理器从 ISR 返回后，中断状态将变更为未激活状态。

② 软件写入清除中断挂起寄存器中的相应位。

- 对于电平中断，如果中断信号仍然有效，则中断状态将保持不变；否则，中断状态将变更为未激活状态。
- 对于脉冲中断，如果状态为挂起或激活，或者状态为激活或挂起，则中断都将 NVIC

紧密结合 Cortex－M 微处理器。当处理器响应一个中断时，NVIC 直接到处理器提供函数的地址去处理中断。因此可以取消一个全局中断处理程序，去查询中断控制器以确定中断源并跳转到相应的中断处理程序，减少了中断的响应时间。

7.1.3 中断优先级

NVIC 允许先处理优先级更高的中断再处理优先级低的中断，允许高优先级中断抢占低优先级的中断处理程序。这有助于减少中断响应时间。并且采用子优先级排序，取代具有 N 位抢占式优先排序，NVIC 可以通过软件配置 N－M 位抢占式优先级和 M 位的子优先级。在这个方案中，两个具有相同的抢占式优先级但不同的子优先级的中断不会引起抢占；尾链的使用使两个中断背靠背的处理。如果两个具有相同优先级的中断（包括子优先级也相同）同时到达，则较低中断号的中断被先行处理；如果中断号仍然相同则排在中断向量表前面的中断先行处理。NVIC 跟踪嵌套的中断处理程序，允许处理器在所有中断嵌套和挂起的中断处理完成后返回。

7.1.4 中断异常

中断异常类型见表 7-1。

表 7-1　异常类型

异常类型	向量号	优先级	向量地址或偏移量	激活
－	0	—	0x0000.0000	复位时栈顶从向量表的首入口加载
复位	1	－3（最高）	0x0000.0004	异步
不可屏蔽的中断（NMI）	2	－2	0x0000.0008	异步
硬故障	3	－1	0x0000.000C	—
存储器管理	4	可编程	0x0000.0010	同步
总线故障	5	可编程	0x0000.0014	精确时同步，不精确时异步
使用故障	6	可编程	0x0000.0018	同步
－	7～10	可编程	－	保留
SVCall（系统服务调用）	11	可编程	0x0000.002C	同步
调试监视器	12	可编程	0x0000.0030	同步
－	13	可编程	－	保留
PendSV（可挂起的系统服务请求）	14	可编程	0x0000.0038	异步
SysTick（系统定时器）	15	可编程	0x0000.003C	异步
中断可编程	16 及其以上	可编程	0x0000.0040 及其以上	异步

7.1.5 寄存器映射

NVIC 寄存器映射见表 7-2。

表 7-2　NVIC 寄存器映射

偏移量	名称	类型	复位	描述
0x100	EN0	R/W	0x0000.0000	中断 0～31 置位使能寄存器
0x104	EN1	R/W	0x0000.0000	中断 32～63 置位使能寄存器

（续）

偏移量	名称	类型	复位	描述
0x108	EN2	R/W	0x0000.0000	中断 64 ~ 95 置位使能寄存器
0x10C	EN3	R/W	0x0000.0000	中断 96 ~ 127 置位使能寄存器
0x110	EN4	R/W	0x0000.0000	中断 128 ~ 138 置位使能寄存器
0x180	DIS0	R/W	0x0000.0000	中断 0 ~ 31 清除使能寄存器
0x184	DIS1	R/W	0x0000.0000	中断 32 ~ 63 清除使能寄存器
0x188	DIS2	R/W	0x0000.0000	中断 64 ~ 95 清除使能寄存器
0x18C	DIS3	R/W	0x0000.0000	中断 96 ~ 127 清除使能寄存器
0x190	DIS4	R/W	0x0000.0000	中断 128 ~ 138 清除使能寄存器
0x200	PEND0	R/W	0x0000.0000	中断 0 ~ 31 置位挂起寄存器
0x204	PEND1	R/W	0x0000.0000	中断 32 ~ 63 置位挂起寄存器
0x208	PEND2	R/W	0x0000.0000	中断 64 ~ 95 置位挂起寄存器
0x20C	PEND3	R/W	0x0000.0000	中断 96 ~ 127 置位挂起寄存器
0x210	PEND4	R/W	0x0000.0000	中断 128 ~ 138 置位挂起寄存器
0x280	UNPEND0	R/W	0x0000.0000	中断 0 ~ 31 清除挂起寄存器
0x284	UNPEND1	R/W	0x0000.0000	中断 32 ~ 63 清除挂起寄存器
0x288	UNPEND2	R/W	0x0000.0000	中断 64 ~ 95 清除挂起寄存器
0x28C	UNPEND3	R/W	0x0000.0000	中断 96 ~ 127 清除挂起寄存器
0x290	UNPEND4	R/W	0x0000.0000	中断 128 ~ 138 清除挂起寄存器
0x300	ACTIVE0	RO	0x0000.0000	中断 0 ~ 31 激活位寄存器
0x304	ACTIVE1	RO	0x0000.0000	中断 32 ~ 63 激活位寄存器
0x308	ACTIVE2	RO	0x0000.0000	中断 64 ~ 95 激活位寄存器
0x30C	ACTIVE3	RO	0x0000.0000	中断 96 ~ 127 激活位寄存器
0x310	ACTIVE4	RO	0x0000.0000	中断 128 ~ 138 激活位寄存器
0x400	PRI0	R/W	0x0000.0000	中断 0 ~ 3 优先级寄存器
0x404	PRI1	R/W	0x0000.0000	中断 4 ~ 7 优先级寄存器
0x408	PRI2	R/W	0x0000.0000	中断 8 ~ 11 优先级寄存器
0x40C	PRI3	R/W	0x0000.0000	中断 12 ~ 15 优先级寄存器
0x410	PRI4	R/W	0x0000.0000	中断 16 ~ 19 优先级寄存器
0x414	PRI5	R/W	0x0000.0000	中断 20 ~ 23 优先级寄存器
0x418	PRI6	R/W	0x0000.0000	中断 24 ~ 27 优先级寄存器
0x41C	PRI7	R/W	0x0000.0000	中断 28 ~ 31 优先级寄存器
0x420	PRI8	R/W	0x0000.0000	中断 32 ~ 35 优先级寄存器
0x424	PRI9	R/W	0x0000.0000	中断 36 ~ 39 优先级寄存器
0x428	PRI10	R/W	0x0000.0000	中断 40 ~ 43 优先级寄存器
0x42C	PRI11	R/W	0x0000.0000	中断 44 ~ 47 优先级寄存器
0x430	PRI12	R/W	0x0000.0000	中断 48 ~ 51 优先级寄存器

（续）

偏 移 量	名 称	类 型	复 位	描 述
0x434	PRI13	R/W	0x0000. 0000	中断 52 ~ 55 优先级寄存器
0x438	PRI14	R/W	0x0000. 0000	中断 56 ~ 59 优先级寄存器
0x43C	PRI15	R/W	0x0000. 0000	中断 60 ~ 63 优先级寄存器
0x440	PRI16	R/W	0x0000. 0000	中断 64 ~ 67 优先级寄存器
0x444	PRI17	R/W	0x0000. 0000	中断 68 ~ 71 优先级寄存器
0x448	PRI18	R/W	0x0000. 0000	中断 72 ~ 75 优先级寄存器
0x44C	PRI19	R/W	0x0000. 0000	中断 76 ~ 79 优先级寄存器
0x450	PRI20	R/W	0x0000. 0000	中断 80 ~ 83 优先级寄存器
0x454	PRI21	R/W	0x0000. 0000	中断 84 ~ 87 优先级寄存器
0x458	PRI22	R/W	0x0000. 0000	中断 88 ~ 91 优先级寄存器
0x45C	PRI23	R/W	0x0000. 0000	中断 92 ~ 95 优先级寄存器
0x460	PRI24	R/W	0x0000. 0000	中断 96 ~ 99 优先级寄存器
0x464	PRI25	R/W	0x0000. 0000	中断 100 ~ 103 优先级寄存器
0x468	PRI26	R/W	0x0000. 0000	中断 104 ~ 107 优先级寄存器
0x46C	PRI27	R/W	0x0000. 0000	中断 108 ~ 111 优先级寄存器
0x470	PRI28	R/W	0x0000. 0000	中断 112 ~ 115 优先级寄存器
0x474	PRI29	R/W	0x0000. 0000	中断 116 ~ 119 优先级寄存器
0x478	PRI30	R/W	0x0000. 0000	中断 120 ~ 123 优先级寄存器
0x47C	PRI31	R/W	0x0000. 0000	中断 124 ~ 127 优先级寄存器
0x480	PRI32	R/W	0x0000. 0000	中断 128 ~ 131 优先级寄存器
0x484	PRI33	R/W	0x0000. 0000	中断 132 ~ 135 优先级寄存器
0x488	PRI34	R/W	0x0000. 0000	中断 136 ~ 138 优先级寄存器
0xF00	SWTRIG	WO	0x0000. 0000	软件触发中断寄存器



7. 2 SysTick 与 NVIC 固件库函数

7. 2. 1 SysTick 固件库

系统定时器的 API 很简单，大致分为两个函数组：配置和使能 SysTick 的常用函数；SysTick 中断处理程序的常用函数。

（1）配置和使能 SysTick 的常用函数

- SysTickEnable( )。
- SysTickDisable( )。
- SysTickPeriodSet( )。
- SysTickPeriodGet( )。

● SysTickValueGet( )。

（2）SysTick 中断处理程序的常用函数

● SysTickIntRegister( )。
● SysTickIntUnregister( )。
● SysTickIntEnable( )。
● SysTickIntDisable( )。

7.2.2 NVIC 固件库

中断控制器 API 函数的主要功能是管理由 NVIC 使用的中断向量表对中断请求的分配。注册中断处理程序就是将其地址插入到中断向量表相应位置即可。注意，如果将未经注册的中断处理程序用来处理中断时，将会产生一个中断错误。因此，只有在处理程序注册之后方可使能中断源，而应在处理程序注销前禁止中断源。

中断控制器的 API 也可大致分为以下 3 个函数组：

① 中断服务函数注册与注销。

② 中断使能与禁止。

③ 中断优先级。

（1）中断服务函数注册与注销的常用函数

● IntRegister( )。
● IntUnregister( )。

（2）中断使能与禁止的常用函数

● IntEnable( )。
● IntDisable( )。
● IntMasterEnable( )。
● IntMasterDisable( )。

（3） 中断优先级的常用函数

● IntPrioritySet( )。
● IntPriorityGet( )。

详细的 NVIC 固件库功能介绍请参考书后附录 E。



7.3 例程

本小节将以中断优先级为例来介绍基于固件的 NVIC 编程与测试方法。

1）中断优先级程序测试接线图如图 7-1 所示。

2）中断优先级程序。

```
//*****************************************************************************
//   文件名:interrupts. c
//   来源:根据 TI 例程改编
//   功能描述:该例介绍了 Cortex - M4 微处理器的 NVIC 中断抢占和尾链功能。嵌套中
//   断是对具有相同优先级、优先级递增和递减的多个中断的合成。当优先级递增时将会
```

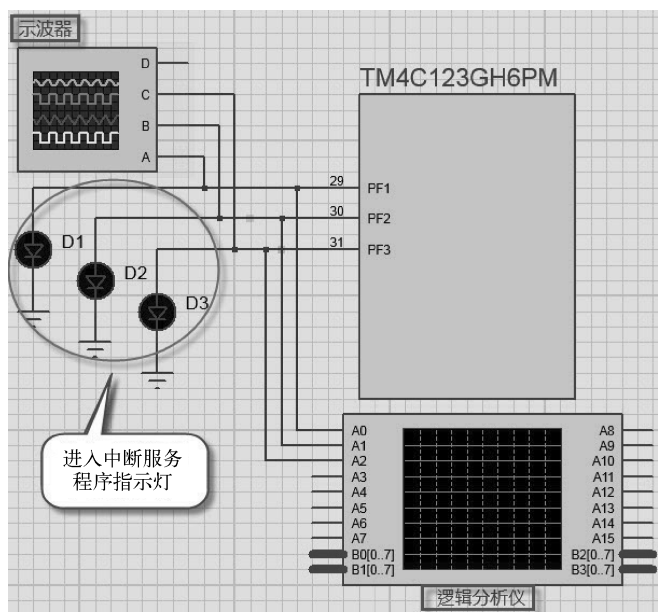


图 7-1　中断优先级程序测试接线图

```
//   产生抢占;而对于相同优先级或递减优先级的情况,中断间会产生背靠背的中断处理
//   (尾链)。当前挂起与执行的中断都会通过 UART 在 Putty 上显示出来,在执行中断服
//   务程序时,不但可在 Putty 上显示其中断状态,而且也可利用 PF1 - PF2 端口外接的板载 LED
//   (RGB)来独立地指示相应的中断状态,并且在退出中断服务程序之前使 LED 熄灭。
//   同时,有条件的读者可以通过如图 7-1 所示的示波器来观测开关时间,或者用逻辑分析
//   仪来观察尾链的速度
// *****************************************************************************

#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_nvic.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/fpu.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/systick.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"

// *****************************************************************************
//接收到的中断计数
// *****************************************************************************
```

```
volatile uint32_t g_ui32Index;

//*****************************************************************************
//当处理 I NT_GPIOA 中断时的 g_ui32Index 值
//*****************************************************************************
volatile uint32_t g_ui32GPIOa;

//*****************************************************************************
//当处理 I NT_GPIOB 中断时的 g_ui32Index 值
//*****************************************************************************
volatile uint32_t g_ui32GPIOb;

//*****************************************************************************
//当处理 I NT_GPIOC 中断时的 g_ui32Index 值
//*****************************************************************************
volatile uint32_t g_ui32GPIOc;

//*****************************************************************************
//发生固件库错误时调用此错误处理例程
//*****************************************************************************
#ifdef DEBUG
void
__error__(char *pcFilename,uint32_t ui32Line)
{
}
#endif

//*****************************************************************************
//延迟指定的秒数
//*****************************************************************************
void
Delay(uint32_t ui32Seconds)
{
    //
    //循环等待
    //
    while(ui32Seconds--)
    {
        //
        //一直等到 SysTick 的值小于 1000
        //
        while(ROM_SysTickValueGet() > 1000)
        {
        }

        //
        //一直等到 SysTick 的值大于 1000
        //
        while(ROM_SysTickValueGet() < 1000)
```

```
            }
        }
    }
}

// ************************************************************************
//通过 UART 在 Putty 上显示中断状态,即显示当前的激活和挂起的中断
// ************************************************************************
void
DisplayIntStatus(void)
{
    uint32_t ui32Temp;

    //
    //显示当前的激活中断
    //
    ui32Temp = HWREG(NVIC_ACTIVE0);
    UARTprintf("\r 激活: %c%c%c ",(ui32Temp & 1) ? '1' : ' ',
              (ui32Temp & 2) ? '2' : ' ',(ui32Temp & 4) ? '3' : ' ');

    //
    //显示当前挂起的中断
    //
    ui32Temp = HWREG(NVIC_PEND0);
    UARTprintf("挂起: %c%c%c",(ui32Temp & 1) ? '1' : ' ',
              (ui32Temp & 2) ? '2' : ' ',(ui32Temp & 4) ? '3' : ' ');
}

// ************************************************************************
//GPIOA 的中断服务程序,这里只简单地保存中断序列号
// ************************************************************************
void
IntGPIOa(void)
{
    //
    //设置 PF1 为高电平来点亮板载的 LED_R,以指示进入该路的中断服务程序
    //
    ROM_GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1,GPIO_PIN_1);

    //
    //将当前的中断状态显示在 Putty 上
    //
    DisplayIntStatus();

    //
    //等待 2 s
    //
    Delay(2);
```

```
    //
    //保存并使其中断序列号增 1
    //
    g_ui32GPIOa = g_ui32Index ++ ;

    //
    //设置 PF1 为低电平使 LED_R 熄灭,以指示退出该路中断服务程序
    //
    ROM_GPIOPinWrite( GPIO_PORTF_BASE,GPIO_PIN_1,0);
}

// ***************************************************************************
//GPIOB 的中断服务程序,即触发 GPIOA 中断和保存其中断序列号
// ***************************************************************************
void
IntGPIOb( void)
{
    //
    //设置 PF2 为高电平来点亮板载的 LED_B,以指示进入该路的中断服务程序
    //
    ROM_GPIOPinWrite( GPIO_PORTF_BASE,GPIO_PIN_2,GPIO_PIN_2);

    //
    //将当前的中断状态显示在 Putty 上
    //
    DisplayIntStatus( );

    //
    //软件触发 GPIOA 中断
    //
    HWREG( NVIC_SW_TRIG) = INT_GPIOA - 16;

    //
    //将当前的中断状态显示在 Putty 上
    //
    DisplayIntStatus( );

    //
    //等待 2 s
    //
    Delay(2);

    //
    //保存并使其中断序列号增 1
    //
    g_ui32GPIOb = g_ui32Index ++ ;

    //
    //设置 PF2 为低电平使 LED_B 熄灭,以指示退出该路中断服务程序
```

```
    //
    ROM_GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2,0);
}

//*****************************************************************************
//GPIOC 的中断服务程序,即触发 GPIOB 中断和保存其中断序列号
//*****************************************************************************
void
IntGPIOc(void)
{
    //
    //设置 PF3 为高电平来点亮板载的 LED_G,以指示进入该路的中断服务程序
    //
    ROM_GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_3,GPIO_PIN_3);

    //
    //将当前的中断状态显示在 Putty 上
    //
    DisplayIntStatus();

    //
    //软件触发 GPIOB 中断
    //
    HWREG(NVIC_SW_TRIG) = INT_GPIOB - 16;

    //
    //将当前的中断状态显示在 Putty 上
    //
    DisplayIntStatus();

    //
    //等待 2 s
    //
    Delay(2);

    //
    //保存并使其中断序列号增 1
    //
    g_ui32GPIOc = g_ui32Index ++;

    //
    //设置 PF3 为低电平使 LED_G 熄灭,以指示退出该路中断服务程序
    //
    ROM_GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_3,0);
}

//*****************************************************************************
//配置 UART 及其引脚,这必须在调用 UARTprintf()函数前完成
//*****************************************************************************
```

```
void
ConfigureUART(void)
{
    //
    //使能在 UART 中使用的 GPIO 外设
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    //
    //使能 UART0
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    //
    //配置 UART 模式中的 GPIO 引脚
    //
    ROM_GPIOPinConfigure(GPIO_PA0_U0RX);
    ROM_GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE,GPIO_PIN_0 | GPIO_PIN_1);

    //
    //使用内部 16 MHz 振荡器作为 UART 时钟源
    //
    UARTClockSourceSet(UART0_BASE,UART_CLOCK_PIOSC);

    //
    //初始化 UART 的控制台 I/O
    //
    UARTStdioConfig(0,115200,16000000);
}

//*****************************************************************************
//主程序。它检查被处理的中断在具有相同优先级、优先级递增和递减时,执行顺序是否正确。
//以及练习中断抢占式优先级和尾链的使用
//*****************************************************************************
int
main(void)
{
    uint32_t ui32Error;

    //
    //使能扩展(lazy)堆栈的中断处理程序
    //这将允许在中断服务程序间使用浮点指令,但需花费额外的堆栈空间
    //
    ROM_FPULazyStackingEnable();

    //
    //设置时钟直接从晶振运行
    //
```

```
ROM_SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_16MHZ);

//
//使能程序中所使用的外设
//
ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

//
//初始化 UART
//
ConfigureUART();

UARTprintf("\033[2J 中断优先级示例\n");

//
//配置 PF1~PF3 为 GPIO 输出以指示中断服务程序的进入/退出
//
ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,GPIO_PIN_1 | GPIO_PIN_2 |
                                          GPIO_PIN_3);
ROM_GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3,0);

//
//设置并使能系统定时器。它将用于中断服务程序中延迟循环的参考。系统定时器的周
//期将设置为 1 s
//
ROM_SysTickPeriodSet(ROM_SysCtlClockGet());
ROM_SysTickEnable();

//
//复位错误指示器
//
ui32Error = 0;

//
//使能中断到处理器
//
ROM_IntMasterEnable();

//
//使能中断
//
ROM_IntEnable(INT_GPIOA);
ROM_IntEnable(INT_GPIOB);
ROM_IntEnable(INT_GPIOC);

//
//显示相同中断优先级的测试从此开始
```

```
//
UARTprintf("\n 相同优先级\n");

//
//设置相同优先级,其中 0x00 为最高优先级
//
ROM_IntPrioritySet(INT_GPIOA,0x00);
ROM_IntPrioritySet(INT_GPIOB,0x00);
ROM_IntPrioritySet(INT_GPIOC,0x00);

//
//复位中断标志
//
g_ui32GPIOa = 0;
g_ui32GPIOb = 0;
g_ui32GPIOc = 0;
g_ui32Index = 1;

//
//软件触发 GPIOC 中断
//
HWREG(NVIC_SW_TRIG) = INT_GPIOC - 16;

//
//将当前的中断状态显示在 Putty 上
//
DisplayIntStatus();

//
//确认以正确的顺序处理中断
//
if((g_ui32GPIOa != 3) || (g_ui32GPIOb != 2) || (g_ui32GPIOc != 1))
{
    ui32Error |= 1;
}

//
//等待 2 s
//
Delay(2);

//
//显示递减中断优先级测试从此开始
//
UARTprintf("\n 递减优先级\n");

//
//设置递减中断优先级(即 C > B > A)
```

```
//
ROM_IntPrioritySet(INT_GPIOA,0x80);
ROM_IntPrioritySet(INT_GPIOB,0x40);
ROM_IntPrioritySet(INT_GPIOC,0x00);

//
//复位中断标志
//
g_ui32GPIOa = 0;
g_ui32GPIOb = 0;
g_ui32GPIOc = 0;
g_ui32Index = 1;

//
//软件触发 GPIOC 中断
//
HWREG(NVIC_SW_TRIG) = INT_GPIOC - 16;

//
//将当前的中断状态显示在 Putty 上
//
DisplayIntStatus();

//
//确认以正确的顺序处理中断
//
if((g_ui32GPIOa != 3) || (g_ui32GPIOb != 2) || (g_ui32GPIOc != 1))
{
    ui32Error |= 2;
}

//
//等待 2 s
//
Delay(2);

//
//显示递增中断优先级测试从此开始
//
UARTprintf("\n 递增优先级\n");

//
//设置递增中断优先级(即 C < B < A)
//
ROM_IntPrioritySet(INT_GPIOA,0x00);
ROM_IntPrioritySet(INT_GPIOB,0x40);
ROM_IntPrioritySet(INT_GPIOC,0x80);
```

```
//
//复位中断标志
//
g_ui32GPIOa = 0;
g_ui32GPIOb = 0;
g_ui32GPIOc = 0;
g_ui32Index = 1;

//
//软件触发 GPIOC 中断
//
HWREG(NVIC_SW_TRIG) = INT_GPIOC - 16;

//
//将当前的中断状态显示在 Putty 上
//
DisplayIntStatus();

//
//确认以正确的顺序处理中断
//
if((g_ui32GPIOa != 1) || (g_ui32GPIOb != 2) || (g_ui32GPIOc != 3))
{
    ui32Error |= 4;
}

//
//等待 2 s
//
Delay(2);

//
//禁止中断
//
ROM_IntDisable(INT_GPIOA);
ROM_IntDisable(INT_GPIOB);
ROM_IntDisable(INT_GPIOC);

//
//禁止处理器中断
//
ROM_IntMasterDisable();

//
//显示测试结果
//
UARTprintf("\nInterrupt Priority =: %s   >: %s   <: %s\n",
          (ui32Error & 1) ?"失败" : "测试通过",
```

```
            (ui32Error & 2) ?"失败" : "测试通过",
            (ui32Error & 4) ?"失败" : "测试通过");

    //
    //无限循环
    //
    while(1)
    {
    }
}
```

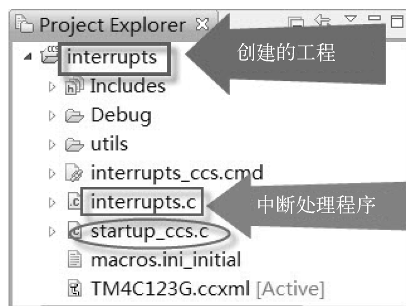


图 7-2　创建的 interrupts 工程

3）在 CCS6 中创建 interrupts 工程，如图 7-2 所示。

4）添加 startup_ccs. c 文件，然后声明中断服务程序（IntGPIOa、IntGPIOb、IntGPIOc）为外部程序，并将其添加到 startup_ccs. c 中断向量表的相应位置，如图 7-3 所示。

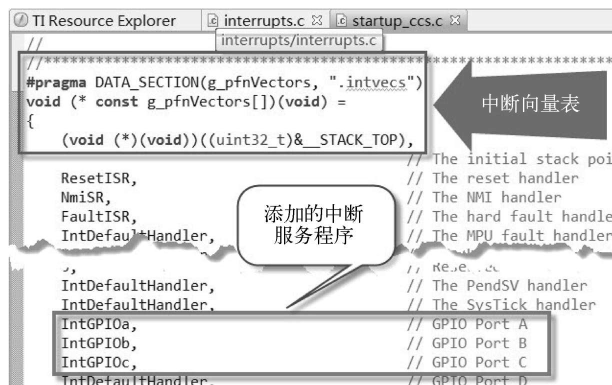


图 7-3　添加中断服务程序到中断向量表中

5）编译 interrupts 工程生成可执行的 . out 格式文件，如图 7-4 所示。

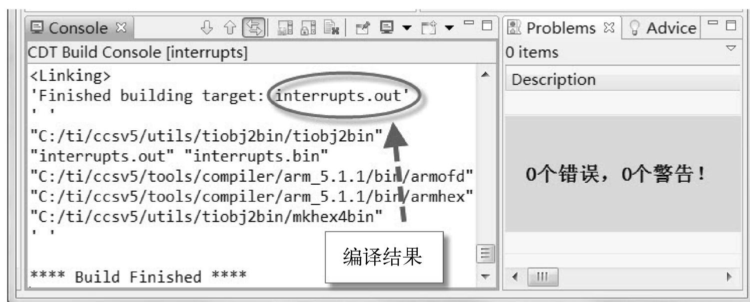


图 7-4　工程编译的结果生成可执行的 . out 文件

6）在 EK－TM4C123GXL 板上对程序进行调试与测试。

① 调试。

a）单击工具栏中如图 7-5 所示的图标及 TM4C123G. ccxml 文件，进入调试视图。

b）首先单击工具栏中的图标连接 EK－TM4C123GXL 板子，然后单击图标导入 .out 文件到板中，如图 7-6 所示。

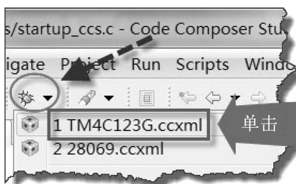


图 7-5　进入调试视图

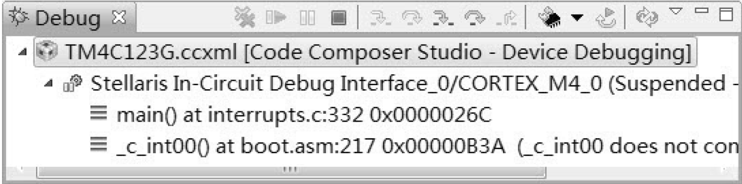


图 7-6　导入编译生成的 .out 文件到 EK－TM4C123GXL 中

c）将变量 g_ui32GPIOa 、g_ui32GPIOb、g_ui32GPIOc 和 g_ui32Index 添加到观察窗口中，如图 7-7 所示。

d）在如图 7-8 所示的位置放置一个断点，接着让程序运行到该断点处，再打开 PuTTY（COM7、波特率设置为 115200Baud）。然后单步执行程序，当执行完 UARTprintf（"\033[2J 中断优先级示例\n"）语句后，在 PuTTY 上出现“中断优先级示例”，说明此前程序运行正确。最后取消该断点。

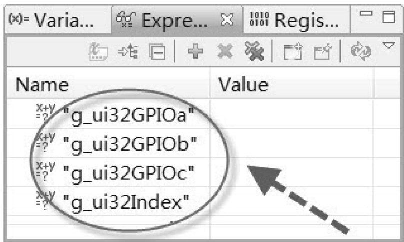


图 7-7　添加到观察窗口中的变量

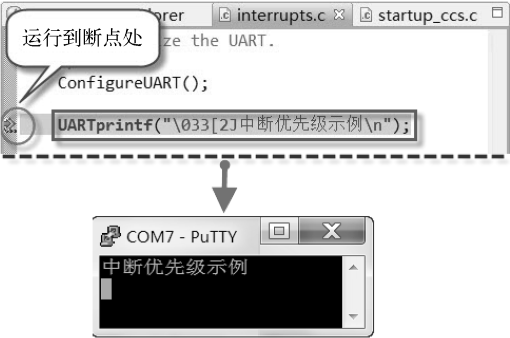


图 7-8　在 UARTprintf. 处放置断点及单步执行结果

e）相同优先级。在 HWREG（NVIC_SW_TRIG）= INT_GPIOC－16 和 GPIOC 中断服务程序中的 ROM_GPIOPinWrite（GPIO_PORTF_BASE，GPIO_PIN_3，GPIO_PIN_3）、DisplayIntStatus（）和 HWREG（NVIC_SW_TRIG）= INT_GPIOB－16 处各设置一个断点。

当程序单步执行完 HWREG（NVIC_SW_TRIG）= INT_GPIOC－16 语句后，触发 GPIOC 中断；在程序单步执行到 DisplayIntStatus（）时，LED_G（绿色 LED）点亮，指示此时已进入 GPIOC 中断服务程序；当单步执行完 DisplayIntStatus（）语句时，在 PuTTY 上的显示序列号为 3（GPIOC 中断服务程序）的中断被激活，如图 7-9 所示。

从图 7-9 的显示与硬件测试结果来看，执行到该断点，激活序列号为 3（GPIOC 中断服务程序）的中断和点亮 LED_G 符合该段程序所表达的思想。注意：此时应删除使用过的所有断点，以便后续设置断点之需。

继续单步运行，当程序执行完 HWREG（NVIC_SW_TRIG）= INT_GPIOB－16 这条语句后，触发 GPIOB 中断。但由于中断 GPIOB 和 GPIOC 为同等优先级，GPIOB 中断不能强占

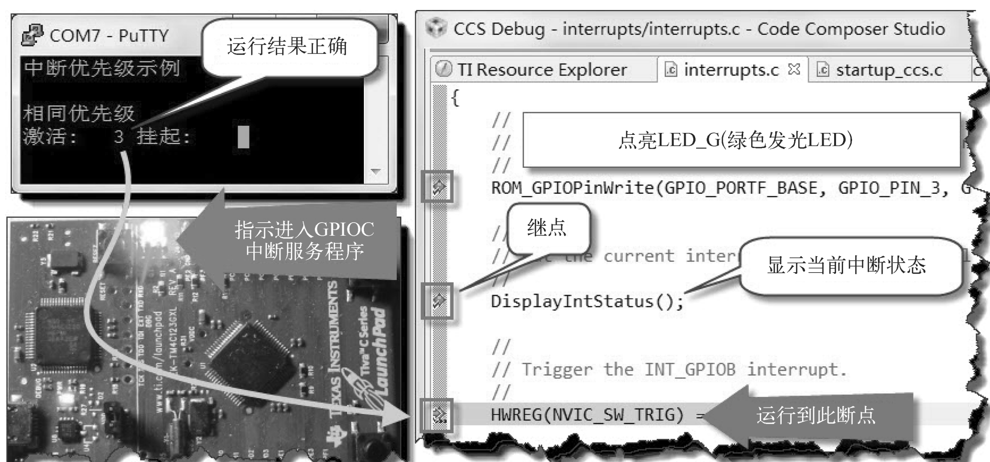


图 7-9　单步执行完 DisplayIntStatus( )后的测试结果

GPIOC 中断的执行权，因此 GPIOB 中断被挂起，而 GPIOC 中断仍处于激活状态，如图 7-10 所示。

当执行完 GPIOC 的中断服务程序时（LED_G 熄灭），GPIOB 中断被激活，继续单步执行指示进入 GPIOB 中断服务程序的蓝色 LED_B 被点亮。这时在 HWREG(NVIC_SW_TRIG) = INT_GPIOA - 16 和 GPIOA 的中断服务程序中的 DisplayIntStatus( ) 处各放置一个断点。当程序执行完 HWREG(NVIC_SW_TRIG) = INT_GPIOA - 16 这条语句后，触发 GPIOA 中断。但由于中断 GPIOA 和 GPIOB 为同等优先级，GPIOA 中断不能强占 GPIOB 中断的执行权，因此 GPIOA 中断被挂起，而 GPIOB 中断仍处于激活状态，如图 7-11 所示。



图 7-10　GPIOC 中断处于激活状态而 GPIOB 中断被挂起



图 7-11　GPIOB 中断处于激活状态而 GPIOA 中断被挂起

当执行完 GPIOB 的中断服务程序时（LED_B 熄灭），GPIOA 中断被激活，继续单步执行指示进入 GPIOA 中断服务程序的红色 LED_R 被点亮，此时在 PuTTY 上显示的中断状态如图 7-12 所示。

f) 递减优先级。首先在 UARTprintf(" \n 递减优先级\n") 处放置一个断点，让程序运行到递减优先级程序段。再按上述放置断点，当程序执行完 HWREG(NVIC_SW_TRIG) = INT_GPIOB - 16 这条语句后，触发 GPIOB 中断。但由于 GPIOB 中断的优先级比 GPIOC 中断低，因此 GPIOB 中断被挂起，而 GPIOC 中断仍处于激活状态，如图 7-13 所示。

当执行完 GPIOC 的中断服务程序时（LED_G 熄灭），GPIOB 中断被激活，继续单步执行指示进入 GPIOB 中断服务程序的蓝色 LED_B 被点亮。

当程序执行完 HWREG(NVIC_SW_TRIG) = INT_GPIOA - 16 这条语句后，触发 GPIOA 中断。但由于 GPIOA 中断的优先级比 GPIOB 中断低，因此 GPIOA 中断被挂起，而 GPIOB

中断仍处于激活状态，如图 7-14 所示。



图 7-12 GPIOA 中断服务程序被激活



图 7-13 GPIOC 中断处于激活状态而 GPIOB 中断被挂起

当执行完 GPIOB 的中断服务程序时（LED_B 熄灭），GPIOA 中断被激活，继续单步执行指示进入 GPIOA 中断服务程序的红色 LED_R 被点亮，此时在 PuTTY 上显示的中断状态如图 7-15 所示。



图 7-14 GPIOB 中断处于激活状态而 GPIOA 中断被挂起



图 7-15 GPIOA 中断服务程序被激活

g）递增优先级。首先在 UARTprintf("\n 递增优先级\n")处放置一个断点，让程序运行到递增优先级程序段。再按上述放置断点，当程序单步执行完 HWREG(NVIC_SW_TRIG) = INT_GPIOC - 16 语句后，触发 GPIOC 中断；在程序单步执行到 DisplayIntStatus()时，LED_G(绿色 LED)点亮，指示此时已进入 GPIOC 中断服务程序，如图 7-16 所示。

当程序执行完 HWREG(NVIC_SW_TRIG) = INT_GPIOB - 16 这条语句后，触发 GPIOB 中断。由于 GPIOB 中断的优先级比 GPIOC 中断高，强占 GPIOC 中断的执行权，转而运行 GPIOB 中断服务程序（绿色 LED_B 被点亮，且此时 LED_G 仍被点亮），如图 7-17 所示。



图 7-16 GPIOC 中断服务程序被激活



图 7-17 GPIOB 中断和 GPIOC 中断均处于激活状态

当程序执行完 HWREG(NVIC_SW_TRIG) = INT_GPIOA - 16 这条语句后，触发 GPIOA 中断。由于 GPIOA 中断的优先级比 GPIOB 中断高，因此强占 GPIOB 中断进入 GPIOA 中断服务程序（红色 LED_R 被点亮，此时 LED_B 和 LED_G 仍被点亮，因此发出白光），如图 7-18 所示。



图 7-18　中断 GPIOA、GPIOB 和 GPIOC 均激活

当执行 GPIOA、GPIOB 和 GPIOC 中断服务程序时，过程如图 7-19 所示。

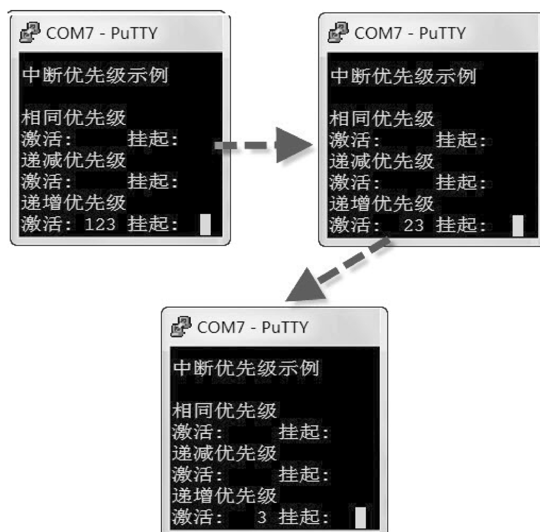


图 7-19　递增优先级中断的执行过程

继续单步执行程序，最后的测试结果如图 7-20 所示。

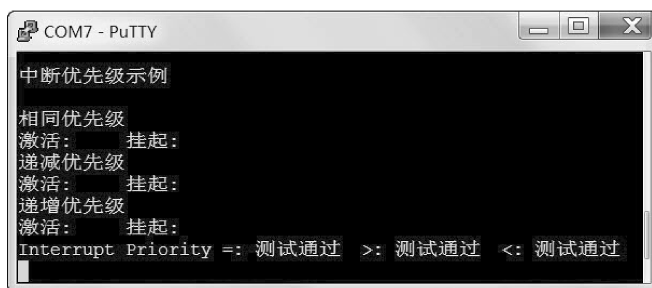


图 7-20　中断优先级示例的测试结果

从以上中断相同优先级、中断递减优先级和递增优先级的单步调试结果来看，程序实现了设计要求达到的效果，对读者加深对中断优先级理解有较好的促进作用。

② 硬件测试。将编译生成的 . out 文件导入到在 EK – TM4C123GXL 板中，打开 PuTTY

观察到的中断执行顺序和板载 LED 的点亮情况，如图 7-21 所示。

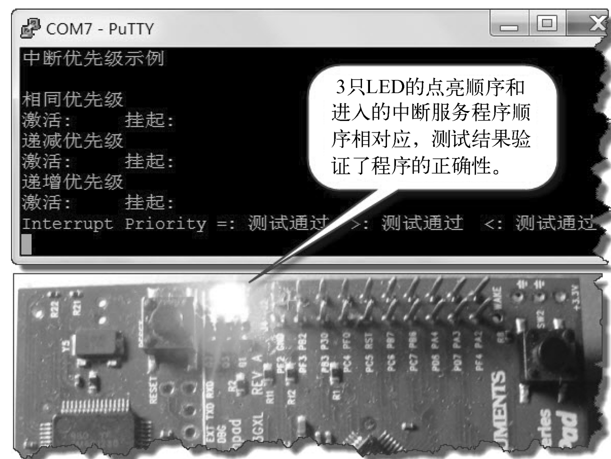


图 7-21　在 EK－TM4C123GXL 板上的测试结果和观察到的现象

在观察相同中断优先级和递减中断优先级时，可以看到这些中断激活和挂起与指示其进入中断服务程序的发光 LED 对应，当被激活的中断服务程序执行结束后马上去执行被挂起的中断，即形成所谓的背对背（尾链）的中断处理。而在观察递增中断优先级时，可以看到 2 只 LED、3 只 LED 同时发光的现象，这是由于其中断服务程序还没有执行完成就被其他优先级高的中断所强占的结果。

第 8 章

内部集成电路接口（I2C）

I2C（Inter-Integrated Circuit）总线通过双线结构（串行数据线 SDA 和串行时钟线 SCL）提供数据的双向传输与连接外部 I2C 设备的接口，例如串行存储器（RAM 和 ROM）、网络设备、LCD、音频发生器等。此外，I2C 总线还能在产品的开发和制造过程中用于系统测试和诊断。TM4C123GH6PM 微控制器提供了 4 个 I2C 模块，能够与总线上的其他 I2C 设备进行读写操作。

I2C 的 API 提供了一组用于使用 Tiva I2C 主从模块的函数。即初始化 I2C 模块、发送和接收数据、获取状态和管理 I2C 模块的中断等。其固件库函数位于 driverlib\i2c.c 目录下，而 driverlib\i2c.h 包含了 I2C API 函数的所有定义。

本章的主要内容：

- I2C 单元
- I2C 固件库函数（见附录 F）
- 例程



8.1 I2C 单元

8.1.1 I2C 特点

TM4C123GH6PM 控制器具有以下特点：

1）I2C 总线上的设备可被指定为主机或从机。

① 同时支持一个主机或从机发送和接收的数据。

② 支持主机和从机同步操作。

2）4 种 I2C 模式。

① 主机传送。

② 主机接收。

③ 从机传送。

④ 从机接收。

3）4 种传输速度。

① 标准（100 kbit/s）。

② 快速（400 kbit/s）。

③ 快速模式 + （1 Mbit/s）。

④ 高速模式（3.33 Mbit/s）。

4） 时钟低电平超时中断。

5） 双从地址功能。

6） 干扰抑制。

7） 主机和从机产生中断。

① 主机在传送或接收数据结束（或由于错误退出）产生中断。

② 从机在主机向其发送数据或发出请求时，或检测到 START 或 STOP 信号时产生中断。

8） 主机带有仲裁和时钟同步功能，支持多主机与 7 位寻址模式。

8.1.2 I2C 模块框图

I2C 模块框图如图 8-1 所示。

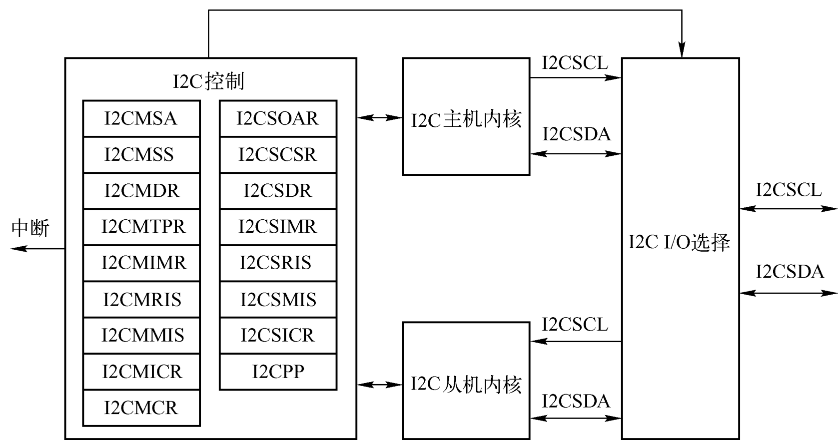


图 8-1 I2C 模块框图

8.1.3 信号描述

I2C 接口的外部信号及其功能描述见表 8-1。

表 8-1 I2C 信号（64LQFP）

引脚名称	引脚编号	引脚复用/引脚赋值	引脚类型	缓冲区类型	描述
I2C0SCL	47	PB2(3)	I/O	OD	I2C 模块 0 时钟
I2C0SDA	48	PB3(3)	I/O	OD	I2C 模块 0 数据
I2C1SCL	23	PA6(3)	I/O	OD	I2C 模块 1 时钟
I2C1SDA	24	PA7(3)	I/O	OD	I2C 模块 1 数据
I2C2SCL	59	PE4(3)	I/O	OD	I2C 模块 2 时钟
I2C2SDA	60	PE5(3)	I/O	OD	I2C 模块 2 数据
I2C3SCL	61	PD0(3)	I/O	OD	I2C 模块 3 时钟
I2C3SDA	62	PD1(3)	I/O	OD	I2C 模块 3 数据

8.1.4 功能描述

每个 I2C 模块都具有主机和从机功能，要使其正常运转，SDA 和 SCL 引脚必须设置成开漏形式。典型的 I2C 总线配置如图 8-2 所示。

1. I2C 总线的功能概述

I2C 总线只使用两个信号：SDA 和 SCL。在 TM4C123GH6PM 微控制器上，它们的名称分别是 I2CSDA 和 I2CSCL。SDA 是双向串行数据线，SCL 是双向串行时钟线。当两根线都为高电平时，总线处于空闲状态。

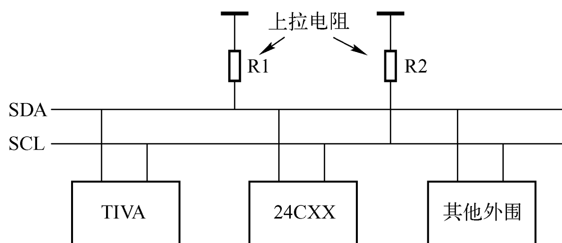


图 8-2　I2C 总线配置

I2C 总线每次传输的数据长度为 9 位，其中包括 8 位数据位和 1 位应答位。每次传输的字节数没有限制，但是每个字节后面必须紧跟 1 位应答位，而且数据传输时必须首先传送最高有效位（MSB）。当接收器不能接收另一个完整的字节时，它会保持时钟线 SCL 为低电平，并强制发送器进入等待状态。在接收器释放时钟线 SCL 之后，数据传送将继续进行。

（1）开始和停止条件

I2C 总线协议定义了两种状态：开始和停止，以便开始和结束数据传输。当 SCL 为高电平时，SDA 线上由高电平变为低电平定义为开始信号；当 SCL 为高电平时，SDA 线上由低电平变为高电平被定义为停止信号。符合开始条件后，总线将占线；达到停止条件后，总线将空闲。如图 8-3 所示。

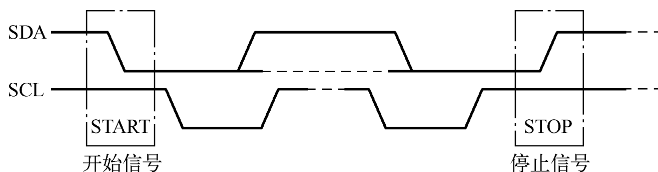


图 8-3　I2C 总线开始和停止条件

STOP 位决定数据周期在结束时是停止还是继续重复运行，直到出现重复的 START 条件。要产生单次传输，应在 I2C 主机从机地址寄存器（I2CMSA）中写入所需的地址，并将 R/S 位清零，在控制寄存器中写入 ACK = X(0 或 1)、STOP = 1、START = 1 与 RUN = 1，来执行单次传输并停止。在传输完成后（或由于错误退出），中断引脚将被激活，数据可从 I2C 主机数据寄存器（I2CMDR）中读出。在 I2C 模块以主接收器模式工作时，ACK 位通常被置位，让 I2C 总线控制器在每个字节接收完成后自动发出一个应答信号。在 I2C 总线控制器无需接收从发送器发送的数据时，该位须清零。

在此模块以从机模式工作时，I2C 从机原始中断状寄存器态（I2CSRIS）中的两位用来监测总线上的开始和停止条件；而 I2C 从机屏蔽中断状态寄存器（I2CSMIS）中的两位将 STARTRIS 与 STOPRIS 位转变成控制器中断（当中断功能已使能）。

（2）带 7 位地址的数据格式

在满足开始条件之后，将发送从机地址。该地址长度为 7 位，第 8 位是数据方向位

（I2CMSA 寄存器中的 R/S 位）。如果 R/S 位被清零，则表示发送操作；如果被置位，则表示接收操作。数据传送由主机产生的停止条件终止。但是，主机无需首先产生停止条件即可与总线上的其他设备通信。因此，在单次传输中可以存在多种接收/发送格式组合，如图 8-4 所示。

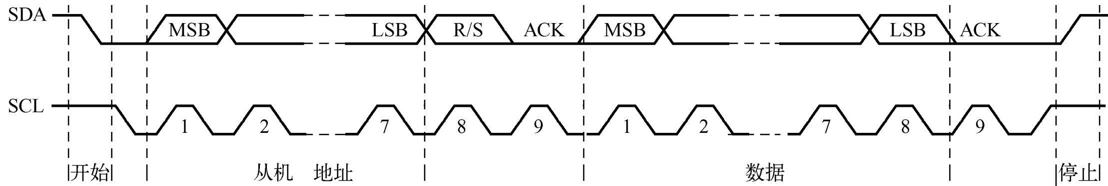


图 8-4　7 位地址的完整数据传输

第一字节中的前 7 位即构成从机地址（如图 8-5 所示）。第 8 位确定报文的方向。若第一字节中的 R/S 位为零，则表示主机向选定的从机发送数据，该位为 1 则表示主机接收从机发送的数据。

（3）数据有效性

SDA 线上的数据在时钟的高电平期间必须稳定，只有在 SCL 为低电平时，数据线才能改变，如图 8-6 所示。

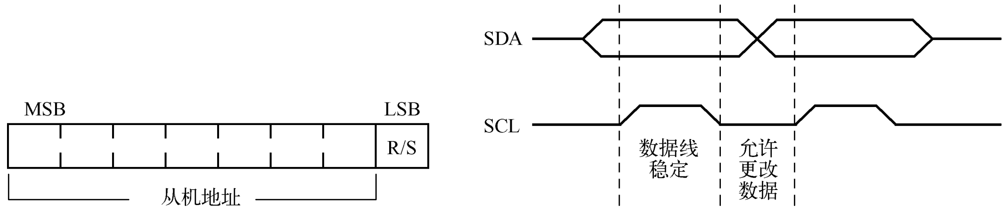


图 8-5　第一字节中的 R/S 位

图 8-6　I2C 总线位传输过程中的数据有效性

（4）应答

所有的总线传输都包含一个由主机产生的应答时钟周期。在应答周期中，发送器（主机或者从机）会释放 SDA 线。在应答时钟周期内，接收器必须将 SDA 拉低。且在应答周期内，接收器发出的数据必须遵循数据有效性要求。

如果从机接收器不应答从机地址，从机必须将 SDA 保持在高电平，以使主机产生停止条件并终止当前的传输。如果主机设备作为接收器，那么它需要应答每一个由从机发起的数据传输。由于主机控制传输中的字节数，通过不应答最后一个字节，主机将数据的末尾告知从机发送器指示数据的结束。接下来从机发送器必须释放 SDA，以便让主机产生停止条件或者重复的开始条件。

（5）仲裁

只有在总线空闲时，主机才可以开始传输。在开始条件的最小保持时间内，可能会有两个或者更多的主机产生开始条件。在这些情况下，当 SCL 为高电平时，SDA 线上将产生仲裁机制。在仲裁过程中，第一个竞争的主机在 SDA 上设置“1”（高电平），而另一个主机发送“0”（低电平），则发送“1”的这个主机将关闭其数据输出阶段并退出，直至总线再次空闲。

仲裁可以在多个位上发生。仲裁的第一个阶段是比较地址位，若两个主机试图寻址相同的设备，则仲裁继续比较数据位。

2. 可用的速度模式

I2C 总线可以在标准模式、快速模式、快速 + 模式和高速模式运行。但选择的模式应和总线上的其他 I2C 器件的速度匹配。

（1）标准、快速和快速 + 模式

选择标准、快速和快速 + 模式，可通过 I2C 主机定时器周期寄存器（I2CMTPR）得到，它们的 SCL 频率分别为 100 kbit/s、400 kbit/s 和 1 Mbit/s。

决定 I2C 时钟速率的参数如下：

```
CLK_PRD        //系统时钟周期
SCL_LP         //SCL 的低电平相位(固定为 6)
SCL_HP         //SCL 的高电平相位(固定为 4)
TIMER_PRD      //I2CMTPR 寄存器的编程值
```

I2C 时钟周期计算公式如下：

$$SCL_PERIOD = 2 \times (1 + TIMER_PRD) \times (SCL_LP + SCL_HP) \times CLK_PRD$$

（2）高速模式

TM4C123GH6PM 的外设 I2C 支持主从高速运行模式，可通过将 I2C 主机控制/状态寄存器（I2CMCS）中的 HS 位置位来配置高速模式。它以 66.6%/33.3% 占空比的高位速率进行数据传输，但通信和仲裁取决于用户所选择的标准，即快速模式和快速 + 模式的速度。当 2CMCS 寄存器中的 HS 位被置位时，当前模式的上拉电阻将被使能。

可使用下面的公式来选择时钟周期，这里 SCL_LP = 2、SCL_HP = 1。

$$SCL_PERIOD = 2 \times (1 + TIMER_PRD) \times (SCL_LP + SCL_HP) \times CLK_PRD$$

当主机操作时，高速模式的数据格式如图 8-7 所示。

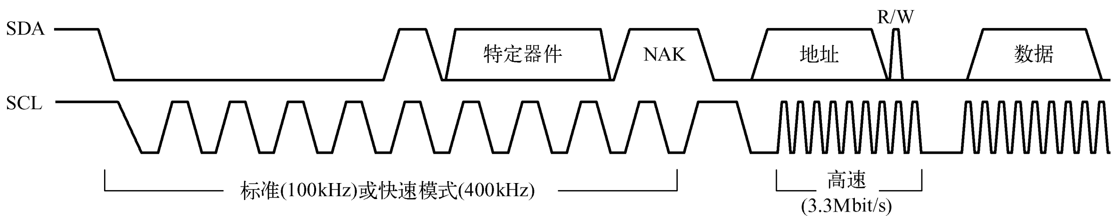


图 8-7　高速模式的数据格式

3. 中断

I2C 将在以下条件时产生中断：

① 主机通信完成。

② 主机仲裁失败。

③ 主机通信错误。

④ 主机总线超时。

⑤ 从机通信接收。

⑥ 从机通信请求。

⑦ 监测到总线上发生停止条件。

⑧ 监测到总线上发生开始条件。

I2C 主机和 I2C 从机模块具有单独的中断信号。尽管在许多条件下两个模块都可以产生中断，但是最后只有一个中断信号将被发送到中断控制器。

（1） I2C 主机中断

当通信结束（发送、接收）、仲裁失败或者通信发生错误等时，I2C 主机模块都会产生一个中断，可通过将 I2CMIMR 寄存器中的 IM 位置位来使能 I2C 主机中断。当满足中断条件时，必须通过软件检查 I2C 主机控制/状态寄存器（I2CMCS）中的 ERROR 和 ARBLST 位，以验证错误并未发生在最后一个通信期间，并且确保没有丢掉仲裁。如果最后一个通信没有得到从机的应答，那么就会发生错误。如果没有错误发生，而且主机也没有丢失仲裁，那么应用程序将继续进行数据传输。当 I2C 主机中断清除寄存器（I2CMICR）中的 IC 位置位时，就可清除该中断。

如果应用程序无需使用中断，则可通过 I2C 主机原始中断状态寄存器（I2CMRIS）随时查看原始中断状态。

（2） I2C 从机中断

在已接收到数据或请求时，从机模块将会产生中断。将 I2C 从机中断屏蔽寄存器（I2CSIMR）中的位 DATAIM 置位即可使能该中断。通过软件检查 I2C 从机控制/状态寄存器（I2CSCSR）中的 RREQ 和 TREQ 位，就可确定该模块应写入（发送）还是读取（接收）来自 I2C 从机数据寄存器（I2CSDR）中的数据。如果从机模块处于接收模式，且已接收第一个字节，那么 FBR 和 RREQ 位将同时置位。如果将 I2C 从机中断清除寄存器（I2CSICR）中的 DATAIC 位置位即可清除该中断。

另外，当监测到开始和停止条件时，从机模块也会产生中断。可通过将 I2C 从机中断屏蔽寄存器（I2CSIMR）中的 STARTIM 和 STOPIM 位置位来使能这些中断，并可将 I2C 从机中断清除寄存器（I2CSICR）中的 STOPIC 和 STARTIC 位置位来清除这些中断。

如果应用程序无需使用中断，则可通过 I2C 从机原始中断状态寄存器（I2CSRIS）随时查看原始中断状态。

4. 回送操作

将 I2C 主机配置寄存器（I2CMCR）中的 LPBK 位置位来让 I2C 模块进入内部回送模式，以进行诊断或者调试工作。在回送模式下，主机和从机发出的 SDA 和 SCL 信号被绑定在一起，用于在无需使用 I/O 接口时也可对器件进行内部测试。

说明：本章没有介绍到的内容请读者参考 TM4C123GH6PM 数据手册的 I2C 部分。



8.2 I2C 固件库函数

8.2.1 主机操作

一般执行步骤如下：

（1） I2CMasterInitExpClk( )

首先调用 I2CMasterInitExpClk( ) 来初始化 I2C 主机模块，设置总线速度和使能主机

模块。

（2）I2CMasterSlaveAddrSet( )

采用 I2CMasterSlaveAddrSet( ) 函数设置从机地址，用于定义传输是一次发送（从主机写数据到从机）还是一次接收（从主机读取从机的数据）。

（3） I2CMasterBusBusy( )

如果连接到一个含有多个主机的 I2C 总线上，Tiva I2C 主机必须在尝试启动传输前，先行调用 I2CMasterBusBusy( ) 来确定是否处于忙状态。

（4）I2CMasterDataPut( )

在确定总线处于非忙碌状态之后，可调用 I2CMasterDataPut( ) 函数来发送数据。

（5） I2CMasterControl

在完成以上步骤后，可调用 I2CMasterControl( ) 函数的以下任一命令来启动总线上的数据传输：

① I2C_MASTER_CMD_SINGLE_SEND。

② I2C_MASTER_CMD_SINGLE_RECEIVE。

③ I2C_MASTER_CMD_BURST_SEND_START。

④ I2C_MASTER_CMD_BURST_RECEIVE_START。

这些命令将会引起总线的主机仲裁、驱动起始序列、发送从机地址与方向位。然后，其余的传输用轮询或中断的方式进行。

（6） I2CMasterErr( )

对于一次发送和接收的情况，轮询方式包括从 I2CMasterBusy( ) 返回的循环。一旦这个函数指示 I2C 主机不再处于忙碌状态，则说明总线传输已经完成，此时可用 I2CMasterErr( ) 来检查错误，以确定数据是否发送完成或已准备好。

（7）I2CMasterDataGet( )

如果没有错误，则说明数据发送完成或已经准备好，这时可用 I2CMasterDataGet( ) 函数来读取数据。

另外，对于突发数据发送和接收的情况，每发送与接收完一个字节，或发送或接收完最后一个字节，轮询方式都会调用 I2CMasterControl( ) 函数。如果在突发传输中检测到错误，则可使用停止命令来调用 I2CMasterControl( ) 函数。

对于中断传输，则需注册一个 I2C 器件的中断处理程序并使能 I2C 主机中断，使之在主机不再忙碌时产生中断。

8.2.2 从机操作

当采用 API 来驱动 I2C 从机模块时，首先需调用 I2CSlaveInit( ) 来初始化 I2C 从机模块，以便使能 I2C 从机模块和初始化从机的地址。在初始化完成后，可调用 I2CSlaveStatus( ) 以轮询从机状态，确认主机是否请求了发送或接收操作。并根据请求操作的类型，调用 I2CSlaveDataPut( ) 或 I2CSlaveDataGet( ) 完成传输。或 I2C 从机采用 I2CIntRegister 注册的一个中断处理程序、使能 I2C 从机中断来处理传输。

8.2.3 I2C 固件库描述

I2C API 被分为三个函数组：

① 中断处理。

② 状态和初始化处理。

③ 发送和接收数据处理。

详细的 I2C 固件库函数功能介绍请参考书后附录 F。



8.3 例程

8.3.1 主从回环例程

为了便于读者利用 EK－TM4C123GXL 开发板来测试 I2C 的程序，本小节将以 TI 的 I2C 回环例程为蓝本，介绍基于固件的 I2C 的编程与测试方法。

1）主从回环例程连线图如图 8-8 所示。

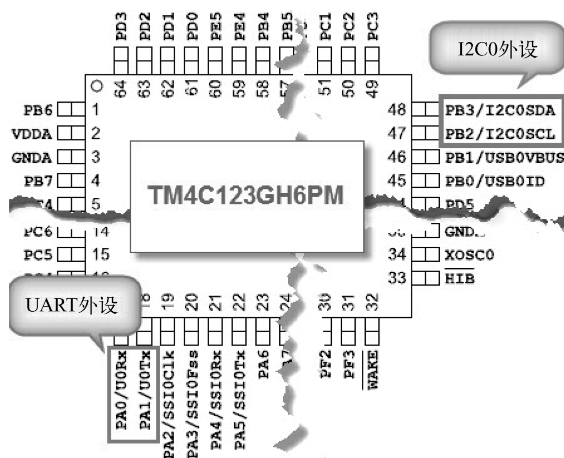


图 8-8　主从回环例程连线图

2）I2C 回环程序的功能说明。

```
//*************************************************************************
//文件名:master_slave_loopback.c
//来源:TI 例程
//!功能描述:
//!这个例子介绍了如何在回环模式下配置 I2C0 模块,以及建立主机与从机模块的数据收发
//!首先设置从机模块地址以便读取主机发送的数据
//!然后将接收到的数据与发送的数据进行对比,以确保收到的数据和发送的数据相同
//!该示例采用轮询方法来发送和接收数据
//!注意:在回环模式下,主机与从机的数据线和时钟线在内部连接在一起
//!本例使用的外设和 I/O 端口如下:
//! - I2C0 外设
//! - GPIO B 端口(I2C0 引脚)
```

```
//! - I2C0SCL:PB2
//! - I2C0SDA:PB3
//!配置下面的 UART 信号仅作显示例程的控制台信息之用:
//! - UART0 外设
//! - GPIO A 端口(UART0 引脚)
//! - UART0RX:PA0
//! - UART0TX:PA1
// ****************************************************************************

#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_i2c.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/gpio.h"
#include "driverlib/i2c.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"

// ****************************************************************************
//要发送的 I2C 数据包的数量
// ****************************************************************************
#define NUM_I2C_DATA 8

// ****************************************************************************
//设置从机模块地址
//带 7 位地址的发送数据格式:[A6:A5:A4:A3:A2:A1:A0:RS]
//若 RS 位为零,则表示主机向选定的从机传输(发送)数据,该位为 1 则表示主机接收从机发送的
//数据
// ****************************************************************************
#define SLAVE_ADDRESS 0x4C

// ****************************************************************************
//初始化控制台函数
// ****************************************************************************
void
InitConsole(void)
{
    //
    //使能 GPIOA 的 UART0 引脚
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    //
    //配置复用引脚 A0 和 A1 为 UART0 功能
    //
    GPIOPinConfigure(GPIO_PA0_U0RX);
```

```
    GPIOPinConfigure(GPIO_PA1_U0TX);

    //
    //使能 UART0 以便可以配置时钟
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    //
    //使用内部 16 MHz 振荡器作为 UART 时钟源
    //
    UARTClockSourceSet(UART0_BASE,UART_CLOCK_PIOSC);

    //
    //选择这些引脚的复用(UART)功能
    //
    GPIOPinTypeUART(GPIO_PORTA_BASE,GPIO_PIN_0 | GPIO_PIN_1);

    //
    //初始化 UART 的控制台 I/O
    //
    UARTStdioConfig(0,115200,16000000);
}

//***************************************************************************
//将 I2C0 配置成主从模式,并使用回环模式使其在内部相连
//***************************************************************************
int
main(void)
{
    uint32_t pui32DataTx[NUM_I2C_DATA];
    uint32_t pui32DataRx[NUM_I2C_DATA];
    uint32_t ui32Index;

    //
    //设置时钟直接从外部晶振/振荡器运行
    //
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_16MHZ);

    //
    //I2C0 外设必须使能后方可使用
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);

    //
    //使能 PortB 端口
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
```

```
//
//将引脚复用端口 B2 和 B3 配置成 I2C0 功能
//
GPIOPinConfigure(GPIO_PB2_I2C0SCL);
GPIOPinConfigure(GPIO_PB3_I2C0SDA);

//
//选择这些引脚为 I2C 功能
//该功能也将 GPIO 引脚配置为 I2C 的操作引脚,并将其设置为开漏操作的弱上拉
//
GPIOPinTypeI2C(GPIO_PORTB_BASE,GPIO_PIN_2 | GPIO_PIN_3);

//
//使能回环模式。回环模式是一个内置功能,可有效地用于 I2C 的调试操作
//它在内部连接 I2C 主机和从机终端,可有效地从主机发送数据、由从机来接收数据
//
HWREG(I2C0_BASE + I2C_O_MCR) | = 0x01;

//
//使能和初始化 I2C0 主模块
//使用 I2C0 模块的系统时钟
//最后一个参数是设置 I2C 的数据速率
//如果为 false,则数据速率设置为 100 kbit/s;如果为 true,则数据速率设置为 400 kbit/s
//
I2CMasterInitExpClk(I2C0_BASE,SysCtlClockGet(),false);

//
//使能 I2C0 从机模块。使能此模块仅用于测试目的
//
I2CSlaveEnable(I2C0_BASE);

//
//设置从机地址为 SLAVE_ADDRESS。在回环模式中,它为发送到 I2CMasterSlaveAddrSet
//函数的一个任意 7 位数(在上面的宏中设置)
//
I2CSlaveInit(I2C0_BASE,SLAVE_ADDRESS);

//
//告诉主机模块,当主机与从机通信时,哪个地址可以放置到总线上。将地址设置为
//SLAVE_ADDRESS(如在从机模块设置)。如果接收参数被设置为 false,则指示将启动 I2C
//主机向从机写入数据;如果为 true,那就指示 I2C 主机启动读取从机发送来的数据
//
I2CMasterSlaveAddrSet(I2C0_BASE,SLAVE_ADDRESS,false);

//
//设置串行控制台用于显示消息
//
InitConsole();
```

```
//
//显示例程中的信息
//
UARTprintf("    I2C    回环示例 ");
UARTprintf("\n    模块 = I2C0");
UARTprintf("\n    模式 = 单发送/接收");
UARTprintf("\n    速率 = 100 kbps\n\n");

//
//初始化要发送的数据
//
pui32DataTx[0] = 'L';
pui32DataTx[1] = 'O';
pui32DataTx[2] = 'O';
pui32DataTx[3] = 'P';

pui32DataTx[4] = 'B';
pui32DataTx[5] = 'A';
pui32DataTx[6] = 'C';
pui32DataTx[7] = 'K';

//
//初始化接收缓冲区
//
for(ui32Index = 0; ui32Index < NUM_I2C_DATA; ui32Index ++)
{
    pui32DataRx[ui32Index] = 0;
}

//
//指示数据的方向
//
UARTprintf("主机向从机发送数据\n");

//
//I2C 主机向从机发送了 8 个数据
//
for(ui32Index = 0; ui32Index < NUM_I2C_DATA; ui32Index ++)
{
    //
    //显示 I2C0 主机正在发送的数据
    //
    UARTprintf("  发送：'%c'  . . .  ",pui32DataTx[ui32Index]);

    //
    //将要发送的数据放置到数据寄存器中
    //
    I2CMasterDataPut(I2C0_BASE,pui32DataTx[ui32Index]);
```

```
        //
        //启动从主机发送数据
        //
        I2CMasterControl(I2C0_BASE,I2C_MASTER_CMD_SINGLE_SEND);

        //
        //等待,直到从机接收完成并应答数据
        //
        while(!(I2CSlaveStatus(I2C0_BASE) & I2C_SLAVE_ACT_RREQ))
        {
        }

        //
        //读取来自从机发送的数据
        //
        pui32DataRx[ui32Index] = I2CSlaveDataGet(I2C0_BASE);

        //
        //等待,直到主机模块完成传输
        //
        while(I2CMasterBusy(I2C0_BASE))
        {
        }

        //
        //显示从机接收到的数据
        //
        UARTprintf("接收:'%c' \n",pui32DataRx[ui32Index]);
    }

    //
    //复位接收缓冲区
    //
    for(ui32Index = 0; ui32Index < NUM_I2C_DATA; ui32Index ++)
    {
        pui32DataRx[ui32Index] = 0;
    }

    //
    //指示数据的方向
    //
    UARTprintf("\n\n从机向主机发送数据\n");

    //
    //将数据的方向变更为 true,使主机读取来自从机发送的数据
    //
    I2CMasterSlaveAddrSet(I2C0_BASE,SLAVE_ADDRESS,true);

    //
```

```
//做一个空接收,以确保不会在第一个接收到的数据中存在无效数据
//
I2CMasterControl(I2C0_BASE,I2C_MASTER_CMD_SINGLE_RECEIVE);

//
//空应答并等待从主机来的接收请求,以清除不应被设置的任何标志
//
while(!(I2CSlaveStatus(I2C0_BASE) & I2C_SLAVE_ACT_TREQ))
{
}

for(ui32Index = 0; ui32Index < NUM_I2C_DATA; ui32Index ++)
{
    //
    //显示 I2C0 从机模块正在发送的数据
    //
    UARTprintf("   发送! %ċ    . . .   ",pui32DataTx[ui32Index]);

    //
    //将要发送的数据放置到数据寄存器中
    //
    I2CSlaveDataPut(I2C0_BASE,pui32DataTx[ui32Index]);

    //
    //告诉主机读取数据
    //
    I2CMasterControl(I2C0_BASE,I2C_MASTER_CMD_SINGLE_RECEIVE);

    //
    //等待,直到从机数据发送完成
    //
    while(!(I2CSlaveStatus(I2C0_BASE) & I2C_SLAVE_ACT_TREQ))
    {
    }

    //
    //从主机读取的数据
    //
    pui32DataRx[ui32Index] = I2CMasterDataGet(I2C0_BASE);

    //
    //显示从机已接收到的数据
    //
    UARTprintf("接收! %ċ \n",pui32DataRx[ui32Index]);
}

//
//告诉用户测试完成
//
```

```
        UARTprintf("\n 完成。\n\n");

        //
        //返回无错误
        //
        return(0);
    }
```

3）在 CCS5.5 中创建 master_slave_loopback 工程，如图 8-9 所示。

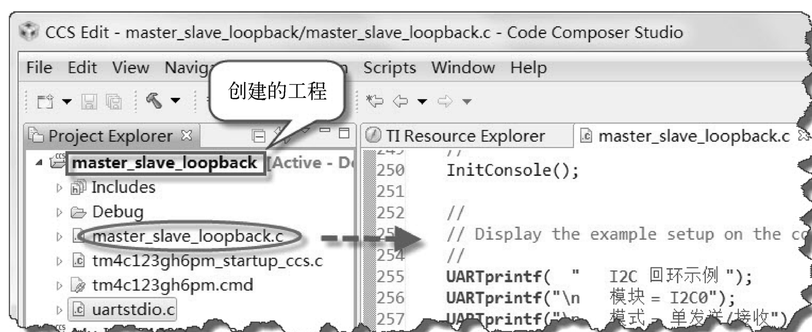


图 8-9　创建的 master_slave_loopback 工程

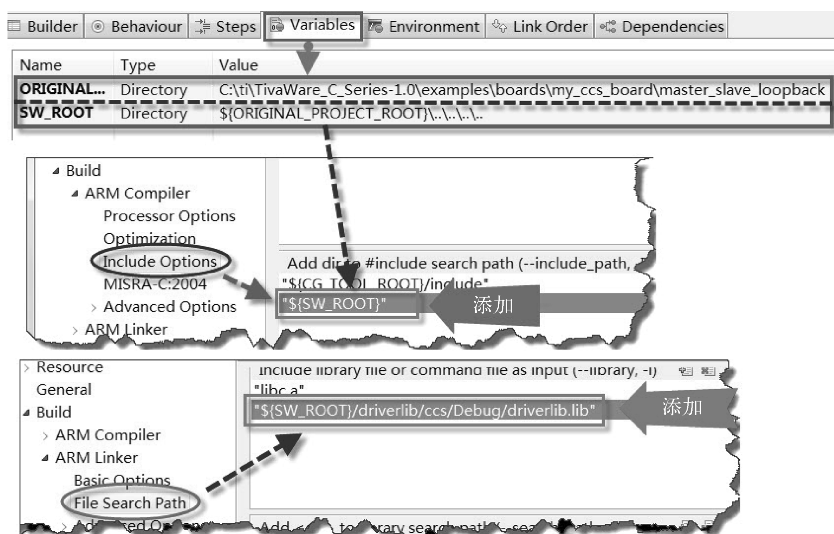


图 8-10　添加包含文件和库文件的搜索路径

4）编译 master_slave_loopback 工程，然后将 .out 文件下载到 EK－TM4C123GXL 板中。

5）打开 PuTTY “串口助手”，程序的运行及测试结果如图 8-11 所示。

从图 8-11 中可以看到，发送的数据和接收到的数据相同，这就验证了以上程序的正确性。



图 8-11　在 EK－TM4C123GXL 板中程序的运行结果

8.3.2　基于 I2C 的 EEPROM 读写例程

1）例程硬件连线图如图 8-12 所示。

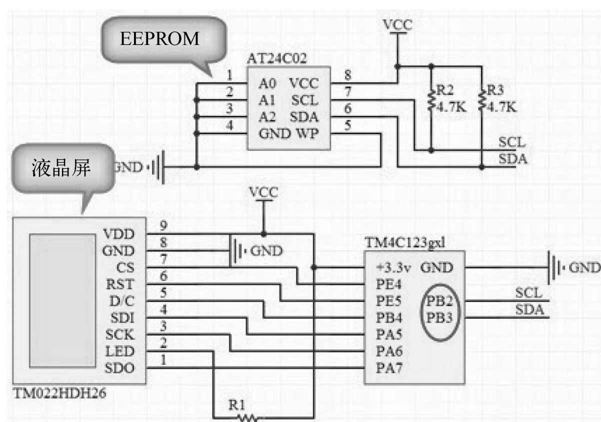


图 8-12　硬件连线图

2）编写 i2c_example. c 程序。

```
// ************************************************************************
//!文件名:i2c_example. c
//!来源:由实验室改编
//!功能描述:通过 tm4c123gh6pm 的硬件 I2C0 单元向 AT24c01(1KB 的 EEPROM)中的地址 0x11,
//!写入"You have learned to use I2C! ",然后再将数据读出,并显示在 LCD 屏幕上
```

```
//!硬件接口:液晶串行通信,接口如下:
//!SDO(MISO) - PA7;       SCK - PA6;       SDI(MOSI) - PA5;
//!DC - PB4;              RESET - PE5;     CS - PE4;
//! AT24C01 的 I2C 接口:  SCL - PB2;       SDA - PB3;
//!注意:1. LCD 采用天马微电子  2.2 寸 TFT 液晶屏 TM022HDH26,驱动芯片为 ILI9341
// ******************************************************************

#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/tm4c123gh6pm.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/i2c.h"

// ******************************************************************
//调用 GPIOPinConfigure()函数输入参数需要声明的文件
// ******************************************************************
#define PART_TM4C123GH6PM
// ******************************************************************
//
//液晶驱动文件,(与 I2C 无关):
//LCD.h 为驱动程序
//font.h 为字符数组文件
//Pin_Def.h 为引脚配置程序
//
// ******************************************************************
#include "LCD.h"
#include "font.h"
#include "Pin_Def.h"

// ******************************************************************
//font.h 中数组声明(与 I2C 无关)
// ******************************************************************
extern uint8_t const image[];
extern uint8_t const hanzi[256];
extern uint8_t const asc2_1608[1520];

// ******************************************************************
//I2C 读写 EEPROM 相关函数声明
// ******************************************************************
#define SLAVE_ADDRESS 0x50
I2CMasterWriteEEPROM(uint8_t *pui8date,uint8_t ui8SlaveAddress,
                     uint8_t ui8EEPAddress,uint32_t ui32Count);
void I2CMasterReadEEPROM(uint8_t *pui8date,uint8_t ui8SlaveAddress,
                         uint8_t ui8EEPAddress,uint32_t ui32Count);
uint8_t inputdate[] = "12345abc";//待传输的字符串
```

```
#ifdef DEBUG
void
__error__(char *pcFilename,unsigned long ulLine)
{
}
#endif

//*********************************************************************
//主函数
//*********************************************************************
int main(void)
{
    //
    //存储从 I2C 读取数据的数组声明
    //
    uint8_t readdata[8];

    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL
_16MHZ);

    //
    //LCD 相关引脚的配置函数,只用 I2C 时,可删除
    //
    GPIOinit();

    //
    //LCD 液晶初始化函数
    //
    Lcd_Init();

    //
    //LCD 屏幕清屏函数
    //BACK_COLOR 设置背景颜色
    //POINT_COLOR 设置画笔颜色
    //详情请参考 LCD.h 文件
    //
    LCD_Clear(WHITE);
    BACK_COLOR = WHITE;
    POINT_COLOR = BLACK;

    //
    //在 10.10 坐标显示字符串"We write to EEPROM:"
    //
    LCD_ShowString(10,10,"We write to EEPROM:");

    //
    //在 10.50 坐标显示将传递的数据:"EEPROM"
    //
```

```
LCD_ShowString(10,50,inputdate);

//
//在 10.90 坐标显示字符串:"initiating..."
//
LCD_ShowString(10,90,"initiating...");

//
//使能 I2C0 模块
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);

//
//使能 GPIOB 模块
//SCL 为 PB2,SDA 为 PB3
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

//
//设置 PB2 为 I2C 的 SCL 功能
//勿忽略添加必要的头文件
//
GPIOPinConfigure(GPIO_PB2_I2C0SCL);

//
//设置 PB3 为 I2C 的 SDA 功能
//勿忽略添加必要的头文件
//
GPIOPinConfigure(GPIO_PB3_I2C0SDA);

//
//进行 PB3(SDA)的引脚配置
//注意:PB3 与 PB2 函数不同,且必须如下设置
//
GPIOPinTypeI2C(GPIO_PORTB_BASE,GPIO_PIN_3);

//
//进行 PB2(SCL)的引脚配置
//注意:PB3 与 PB2 函数不同,且必须如下设置
//因为 SCL 引脚与 SDA 不同,不能设置为开漏模式
//
GPIOPinTypeI2CSCL(GPIO_PORTB_BASE,GPIO_PIN_2);

//
//配置 I2C0 模块的通信时钟,false 代表传输速率为 100 kbit/s
//true 代表速率为 400 kbit/s
//
I2CMasterInitExpClk(I2C0_BASE,SysCtlClockGet(),false);
```

```
    //
    //在 10.90 坐标显示字符串:"writing...        "
    //
    LCD_ShowString(10,90,"writing...        ");

    //
    //将数组 inputdata[ ]的数据通过 I2C 写入
    //SLAVE_ADDRESS(0x50)地址的 EEPROM 中
    //写入的寄存器地址为 0X00,写入数据长度为 8
    //
    I2cMasterWriteEEPROM(inputdate,SLAVE_ADDRESS,0x00,8);

    //
    //在 10.90 坐标显示字符串:"Reading...        "
    //
    LCD_ShowString(10,90,"Reading...        ");

    //
    //通过 I2C 将 EEPROM 中数据写入数组 readdata[ ]
    //从机地址为 SLAVE_ADDRESS(0x50)
    //读取的寄存器地址为 0X00,读取数据长度为 8
    //
    I2cMasterReadEEPROM(readdata,SLAVE_ADDRESS,0x00,8);

    //
    //在 10.90 坐标显示字符串:"We read from EEPROM:"
    //
    LCD_ShowString(10,90,"We read from EEPROM:");

    //
    //在 10.130 坐标显示读取的数据
    //
    LCD_ShowString(10,130,readdata);

    return 0;
}

//*****************************************************************
//函数名称:I2CMasterWriteEEPROM
//函数功能:将数据通过 I2C0 模块写入 AT24C01C 的 EEPRom 中
//输入参数:
//              *pui8date          需写入的数据
//              ui8SlaveAddress    从机地址
//              ui8EEPAddress      EEPRom 中的存储地址
//              ui32Count          要写入的数据的长度
//                                 AT24C01 一次最多可连续写入 8B
//输出参数:无
//*****************************************************************
I2cMasterWriteEEPROM(uint8_t *pui8date,uint8_t ui8SlaveAddress,
```

```
                        uint8_t ui8EEPAddress,uint32_t ui32Count)
{
    uint32_t DataNumber;                    //循环变量

    uint32_t MsterCMD;                      //传输状态暂存变量

    //
    //设置主机模式下从机的地址为 ui8SlaveAddress
    //读写位为写(false),读(true)
    //
    I2CMasterSlaveAddrSet(I2C0_BASE,ui8SlaveAddress,false);

    //
    //将 EEPRom 地址写入发送寄存器
    //
    I2CMasterDataPut(I2C0_BASE,ui8EEPAddress);

    //
    //写入 EEPRom 存储地址
    //本质上是主机发送从机地址并传输一个字节数据
    //但不发送 I2C 结束标志
    //
    I2CMasterControl(I2C0_BASE,I2C_MASTER_CMD_BURST_SEND_START);

    //
    //检测总线是否忙,即等待传输完毕
    //
    while(I2CMasterBusy(I2C0_BASE));

    //
    //检测 I2C 传输是否出错
    //
    while(!(I2CMasterErr(I2C0_BASE) == I2C_MASTER_ERR_NONE));

    //
    //进入数据发送循环
    //备注:写入数据的第一个字节是 I2C 传输的第二个字节数据
    //
    for(DataNumber = 0; DataNumber < ui32Count; DataNumber ++ )
    {
        if(DataNumber == (ui32Count - 1))
        {
            //
            //当剩下一个字节需要发送时,指令变为 I2C_MASTER_CMD_BURST_SEND_FINISH
            //
            MsterCMD = I2C_MASTER_CMD_BURST_SEND_FINISH;
        }
        else
        {
```

```
            //
            //发送的不是最后一个字节
            //指令变为 I2C_MASTER_CMD_BURST_SEND_CONT
            //
            MsterCMD = I2C_MASTER_CMD_BURST_SEND_CONT;
        }

        //
        //将需要发送的 1 个字节数据写入 I2C 发送寄存器(MDR)
        //
        I2CMasterDataPut(I2C0_BASE, * pui8date);

        //
        //启动数据发送
        //
        I2CMasterControl(I2C0_BASE,MsterCMD);

        //
        //检测总线是否忙,即等待传输完毕
        //
        while(I2CMasterBusy(I2C0_BASE));

        //
        //检测 I2C 传输是否出错
        //
        while(!(I2CMasterErr(I2C0_BASE) == I2C_MASTER_ERR_NONE));

        //
        //指针加 1,移动到下一个需要发送的字节
        //
        pui8date ++;
    }
}


// ************************************************************************
//函数名称:I2cMasterReadEEPROM
//函数功能:通过 I2C0 模块读取 AT24C01C 的 EEPRom 中的数据
//          根据 AT24C01C 要求,要先写入存储地址,然后读取数据
//          所以整个过程需要先写后读
//输入参数:
// *             pui8date          读出的数据存放的指针变量
//               ui8SlaveAddress   从机地址
//               ui8EEPAddress     EEPRom 中的存储地址
//               ui32Count         要读取的数据的长度
//                                 AT24C01 一次最多可连续读取 8B
//输出参数:无
// ************************************************************************
void I2cMasterReadEEPROM(uint8_t  * pui8date,uint8_t ui8SlaveAddress,
```

```
                        uint8_t ui8EEPAddress,uint32_t ui32Count)
{
    uint32_t DataNumber;            //循环变量

    uint32_t MsterCMD;              //传输状态暂存变量

    //
    //设置主机模式下从机的地址为 ui8SlaveAddress,读写位为写(false),读(true)
    //
    I2CMasterSlaveAddrSet(I2C0_BASE,ui8SlaveAddress,false);

    //
    //将 EEPRom 地址写入发送寄存器
    //
    I2CMasterDataPut(I2C0_BASE,ui8EEPAddress);

    //
    //写入 EEPRom 存储地址
    //本质上是主机发送从机地址并传输一个字节数据
    //此次发送 I2C 结束标志
    //
    I2CMasterControl(I2C0_BASE,I2C_MASTER_CMD_SINGLE_SEND);

    //
    //检测总线是否忙,即等待传输完毕
    //
    while(I2CMasterBusy(I2C0_BASE));

    //
    //检测 I2C 传输是否出错
    //
    while(!(I2CMasterErr(I2C0_BASE) == I2C_MASTER_ERR_NONE));

    //
    //设置主机模式下从机的地址为 ui8SlaveAddress
    //读写位为读(true)
    //
    I2CMasterSlaveAddrSet(I2C0_BASE,ui8SlaveAddress,true);

    //
    //进入数据接受循环
    //
    for(DataNumber = 0; DataNumber < ui32Count; DataNumber ++ )
    {
        if(ui32Count == 1)
        {
            //
            //如果接受长度只有 1 个字节
            //指令为 I2C_MASTER_CMD_SINGLE_RECEIVE
```

```
        //
        MsterCMD = I2C_MASTER_CMD_SINGLE_RECEIVE;
    }
    else
    {
        if(DataNumber == 0)
        {
            //
            //接收第一个字节
            //指令为 I2C_MASTER_CMD_BURST_RECEIVE_START
            //
            MsterCMD = I2C_MASTER_CMD_BURST_RECEIVE_START;
        }
        else if(DataNumber == (ui32Count - 1))
        {
            //
            //接收最后一个字节
            //指令为 I2C_MASTER_CMD_BURST_RECEIVE_FINISH
            //
            MsterCMD = I2C_MASTER_CMD_BURST_RECEIVE_FINISH;
        }
        else
        {
            //
            //既不是第一个也不是最后一个字节
            //指令为 I2C_MASTER_CMD_BURST_RECEIVE_CONT
            //
            MsterCMD = I2C_MASTER_CMD_BURST_RECEIVE_CONT;
        }
    }

    //
    //开始发送数据
    //
    I2CMasterControl(I2C0_BASE,MsterCMD);

    //
    //检测总线是否忙,即等待传输完毕
    //
    while(I2CMasterBusy(I2C0_BASE));

    //
    //检测 I2C 传输是否出错
    //
    while(!(I2CMasterErr(I2C0_BASE) == I2C_MASTER_ERR_NONE));

    //
    //将主机数据寄存器(MDR)中的数据读出
    //并存储到指定数组
```

```
            //
            * pui8date = I2CMasterDataGet( I2C0_BASE) ;

            //
            //存储数据的指针加 1
            //
            pui8date ++ ;
        }
    }
```

3）在 LaunchPad 板子测试。

① 建立工程、编译、下载等，过程省略。

② 在如图 8-13 处设置一个断点（即读 EEPROM 函数处）。



图 8-13　在读 EEPROM 函数处设置一个断点

③ 单击运行按钮，在断点处的运行结果如图 8-14 所示。

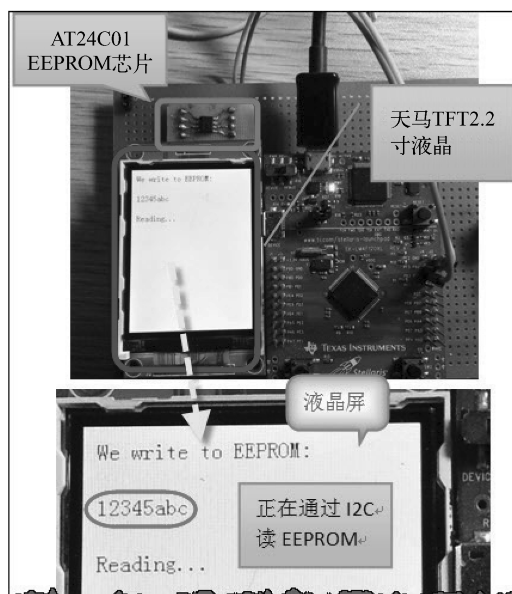


图 8-14　在断点（读 EEPROM 函数处）的运行结果

④ 然后全速运行，读写程序的运行结果如图 8-15 所示。

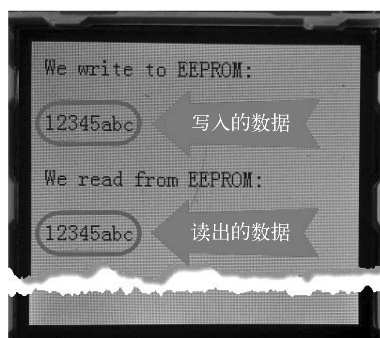


图 8-15　通过 I2C 模块写入的数据和读出的数据比较

从图 8-15 中可以看到，写入的数据和读出的数据一致，验证了上述 I2C 读写程序的正确性。

说明：没有 M4/M3 板卡的读者可参考 soft_i2c_atmel 例程（配套资源第 8 章给出），将芯片选为 LM3S316 等，即可在 Proteus 中进行测试。该程序 TI 写得比较有特色，它把一些可能在读/写中出现的情况作为一种状态，并以状态机的方式来读/写 I2C_EEPROM 中的数据，值得读者借鉴（与经常在 STM32 或 C51 单片机中使用的写法有所不同）。

第 9 章

同步串行接口（SSI）

同步串行接口（Synchronous Serial Interface，SSI）是一种常用的工业通信接口，由摩托罗拉公司发布被多家国际大公司采用。市面上一些流行的电子设备为了减少信号线的数量，降低成本多采用 SPI 接口，比如 SD 卡、液晶屏、CAN 等。TM4C123GH6PM 微控制器包含 4 个同步串行接口（SSI）模块。每个 SSI 都可使用主机或从机接口与外设进行同步串行通信。此外，TM4C123GH6PM 还支持飞思卡尔的 SPI 与 MICROWIRE 传输。对于包含 DMA 控制器的器件，SSI 模块提供了 DMA 接口以使设备通过 DMA 传输数据。

该驱动程序包含在 driverlib/ssi. c 中，driverlib/ssi. h 包含应用程序使用的 API 定义。

本章的主要内容：

- SSI 模块
- SSI 固件库函数（见附录 G）
- 例程



9. 1　SSI 单元

9. 1. 1　SSI 的特点

TM4C123GH6PM SSI 单元的特点如下：

1）提供可编程的控制接口，能与 Freescale 的 SPI 接口、MICROWIRE 或 TI 的同步串行接口相连。

2）主机或从机的工作模式。

3）可编程的时钟位速率与预分频器。

4）独立的发送 FIFO 和接收 FIFO，均为 16 位宽和 8 个单元深。

5）从 4 ~16 位可编程的数据帧大小。

6）内部环回测试模式用于诊断/调试。

7）标准 FIFO 中断与传输结束中断。

8）使用微直接内存访问控制器（μDMA）进行高效数据传输：

① 独立的发送和接收通道。

② 在接收 FIFO 中有数据时将产生单次请求；当接收 FIFO 中包含 4 个数据单元时会产

生突发请求。

③ 在发送 FIFO 中有空闲单元时将产生单次请求；当发送 FIFO 中包含 4 个空闲单元时会产生突发请求。

9.1.2 模块框图

同步串行接口（SSI）单元的功能模块框图如图 9-1 所示。

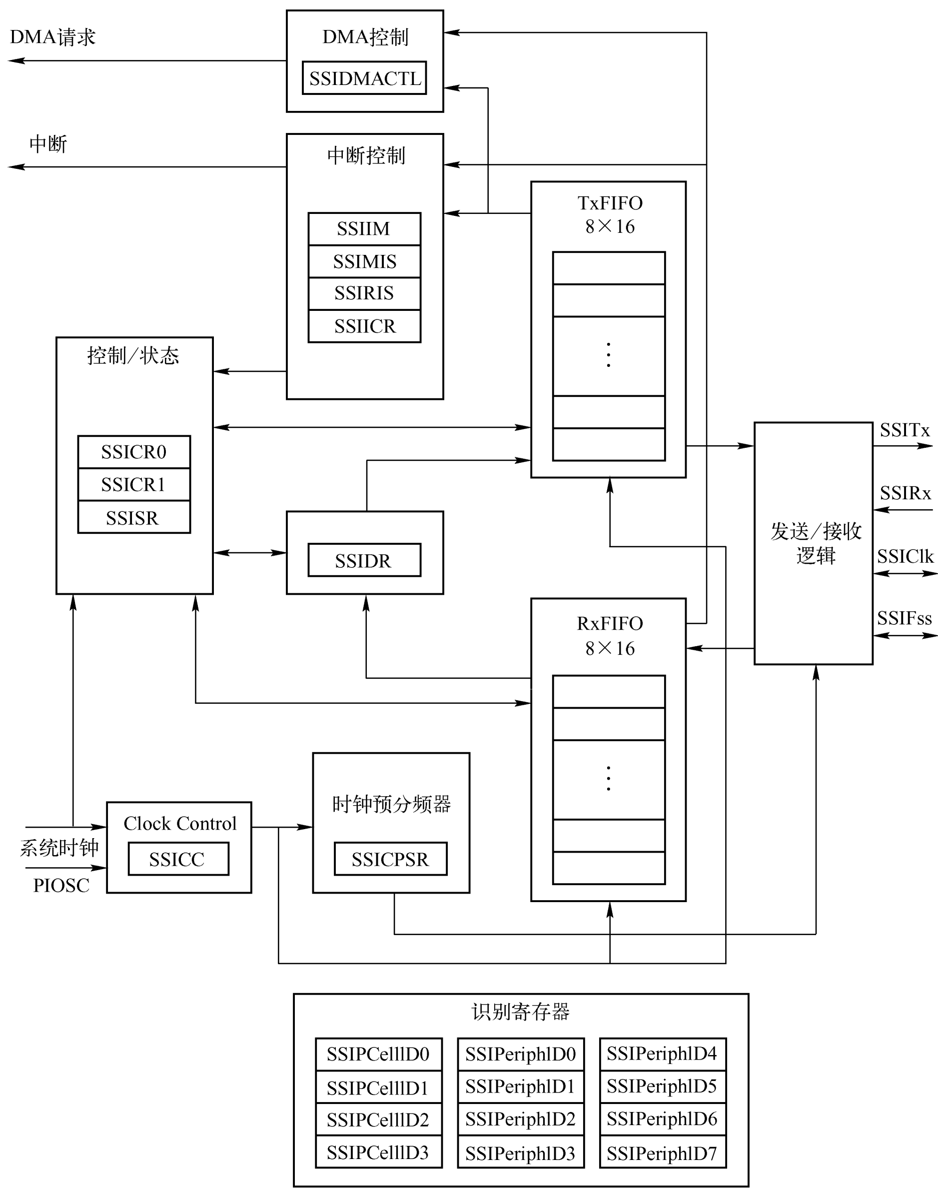


图 9-1　SSI 单元的功能模块框图

9.1.3 信号描述

表 9-1 列出了 SSI 模块的外部信号，并描述了每个信号的功能。

表 9-1　SSI 信号（64LQFP）

引脚名称	引脚编号	引脚复用/赋值	引脚类型	缓冲区类型	描述
SSI0Clk	19	PA2(2)	I/O	TTL	SSI 模块 0 时钟信号
SSI0Fss	20	PA3(2)	I/O	TTL	SSI 模块 0 帧信号
SSI0Rx	21	PA4(2)	I	TTL	SSI 模块 0 接收信号
SSI0Tx	22	PA5(2)	O	TTL	SSI 模块 0 发送信号
SSI1Clk	30 61	PF2(2) PD0(2)	I/O	TTL	SSI 模块 1 时钟信号
SSI1Fss	31 62	PF3(2) PD1(2)	I/O	TTL	SSI 模块 1 帧信号
SSI1Rx	28 63	PF0(2) PD2(2)	I	TTL	SSI 模块 1 接收信号
SSI1Tx	29 64	PF1(2) PD3(2)	O	TTL	SSI 模块 1 发送信号
SSI2Clk	58	PB4(2)	I/O	TTL	SSI 模块 2 时钟信号
SSI2Fss	57	PB5(2)	I/O	TTL	SSI 模块 2 帧信号
SSI2Rx	1	PB6(2)	I	TTL	SSI 模块 2 接收信号
SSI2Tx	4	PB7(2)	O	TTL	SSI 模块 2 发送信号
SSI3Clk	61	PD0(1)	I/O	TTL	SSI 模块 3 时钟信号
SSI3Fss	62	PD1(1)	I/O	TTL	SSI 模块 3 帧信号
SSI3Rx	63	PD2(1)	I	TTL	SSI 模块 3 接收信号
SSI3Tx	64	PD3(1)	O	TTL	SSI 模块 3 发送信号

9.1.4 功能简介

SSI 对外设接收的数据进行串→并转换。CPU 可访问数据、控制与状态信息。发送/接收通道都内置了 FIFO 存储器，在发送与接收模式下最多可支持存储 8 个 16 位数。另外，SSI 还支持 μDMA 接口，它可将发送和接收 FIFO 配置成 μDMA 模块的目的/源地址。

1. 位速率的产生

SSI 模块内置可编程的位速率时钟分频器与预分频器来产生串行输出时钟。SSI 模块支持 2 MHz 及更高的位速率，但最高位速率由外设决定。

串行位速率由输入时钟（SysClk）分频后得到。首先，使用 2～254 之间的偶数分频值 CPSDVSR 对输入时钟进行分频，它可在 SSI 时钟预分频寄存器（SSICPSR）中进行配置。而后再使用 1～256 之间的一个数（即 1+SCR）对时钟进一步分频，这里的 SCR 在 SSI 控制 0 寄存器（SSICR0）中配置。

输出时钟 SSIClk 的频率定义如下：

$$SSIClk = SysClk / (CPSDVSR * (1 + SCR))$$

注意：在主机模式下，系统时钟必须至少为 SSIClk 的两倍，而 SSIClk 不能超过 25 MHz。在从机模式下，系统时钟 必须至少为 SSIClk 的 12 倍，而 SSIClk 不能超过 6.67 MHz。

2. FIFO 操作

（1）发送 FIFO

一般发送 的 FIFO 是一组 16 位宽、8 单元深的先入先出缓冲区。CPU 通过写 SSI 数据寄存器（SSIDR）将数据写入发送 FIFO 中，且数据在由发送逻辑读出之前一直保存在发送缓冲区中。

当 SSI 配置为主机或从机时，并行数据将进行并→串转换，然后在通过 SSITx 引脚发送给相应的从机或主机之前，先将这些数据写入到发送 FIFO 中。

当工作于从机模式时，SSI 模块在每次主机启动通信后才发送数据。若发送 FIFO 为空，则在主机启动会话时 SSI 模块将把发送到 FIFO 中的最早一个数据发送出去。假设从机启动 SSI 模块时钟以来，写入到发送 FIFO 中的有效数据不足 8 位，则 SSI 模块将发送 0。因此，FIFO 中必须包含按照会话要求的有效数据。SSI 模块可以在发送 FIFO 为空时产生中断或 μDMA 请求。

（2）接收 FIFO

一般 SSI 接收 FIFO 是一组 16 位宽、8 单元深的先入先出缓冲区。从串行接口接收到的数据在由 CPU 读出之前一直保存在缓冲区中，CPU 通过读 SSIDR 寄存器来访问读入 FIFO 的数据。在 SSI 配置为主机或从机时，首先将从 SSIRx 引脚接收到的串行数据保存起来，然后并行加载到相应的从机或主机接收 FIFO 中。

3. 中断

SSI 将在出现下列情况时产生中断：

① 发送 FIFO 服务（发送 FIFO 半满或更低）。

② 接收 FIFO 服务（接收 FIFO 半满或更高）。

③ 接收 FIFO 超时。

④ 接收 FIFO 溢出。

⑤ 传输结束。

⑥ 接收 DMA 传输完成。

⑦ 发送 DMA 传输完成。

在发送给中断控制器之前，所有中断事件先进行一次逻辑或操作，因此同一时刻不管实际发生了多少个 SSI 中断事件，SSI 模块都只向中断控制器发出一个中断请求。将 SSI 中断屏蔽寄存器（SSIIM）中相应的位清零来单独屏蔽这 4 个中断中的一个。将相应的屏蔽位置位来使能中断。

SSI 模块不仅提供组合的中断输出，而且还分别提供各个中断源的输出，所以在处理中断时既可采用全局中断处理子程序，也可采用模块化的设备驱动程序。动态的发送/接收数据流中断与静态的状态中断相互独立，便于即时响应 FIFO 触发深度进行读写操作。若想了解独立中断源的状态，可查询 SSI 原始中断状态寄存器（SSIRIS）和 SSI 屏蔽中断状态寄存器（SSIMIS）。

接收 FIFO 设有 32 个 SSIClk 时钟周期的超时周期，只要接收 FIFO 从空状态变为非空状态将开启超时周期。若 RXFIFO 在后续的 32 个时钟内再次变为空状态，才会将其

中止并复位。则中断服处理程序应在读取接收 FIFO 之后及时将 SSI 中断清除寄存器（SSIICR）中的 RTIC 位置位，以清除接收 FIFO 超时中断。并且该清除操作不能太迟执行，否则有可能造成中断处理子程序在中断被清除之前已返回，以及不必要地重复进入中断等问题。

传输结束（EOT）中断表示数据已经传输完成。该中断可以用来指示何时禁止 SSI 模块的时钟或进入休眠模式。另外，因数据的发送和接收是同时完成的，该中断也可用于实时指示接收 FIFO 中的数据已经就绪，无需等待接收 FIFO 超时。

4. 帧格式

SSI 的数据帧长度从 4～6 位可编程，并且始终按照高位在前的顺序传输。有三种基本的帧类型可供选择：

① TI 同步串行。

② 飞思卡尔 SPI。

③ MICROWIRE。

对于上述三种帧格式，串行时钟（SSIClk）在 SSI 空闲时保持非激活状态，只有当数据的发送或接收处于激活状态时，SSIClk 才可在所设置的频率下工作。利用 SSIClk 的空闲状态可提供接收超时指示，在一个超时周期后，如果接收 FIFO 中仍含有数据，则会产生超时指示。

对于 Freescale SPI 和 MICROWIRE 这两种帧格式，在串行帧（SSIFss）引脚为低电平时有效，并在整个帧的传输过程中一直保持有效。

而对于 TI 的同步串行帧格式，在发送每个帧前，SSIFss 引脚将发出一个以上升沿开始并持续一个时钟周期的脉冲。在该帧格式中，SSI 和片外从设备将在 SSIClk 的上升沿驱动各自的数据输出，并在下降沿锁存另一个器件的数据。

与 TI 和 Freescale 的全双工帧格式不同，MICROWIRE 采用半双工方式，使用特殊的主－从传输技术。在该模式中，当帧开始传输时将向外设从设备发送一个 8 位的控制报文。并且在发送控制字期间不会接收数据。当报文发送结束后，外设从设备将立即对报文进行译码，发送完 8 位报文的最后一位后，在该从设备需等待 1 个串行时钟周期后，方可响应请求的数据应答。返回的数据长度为 4～16 位，使总的帧长为 13～25 位。

（1）TI 的同步串行帧格式

TI 同步串行帧格式的单次传输如图 9-2 所示。

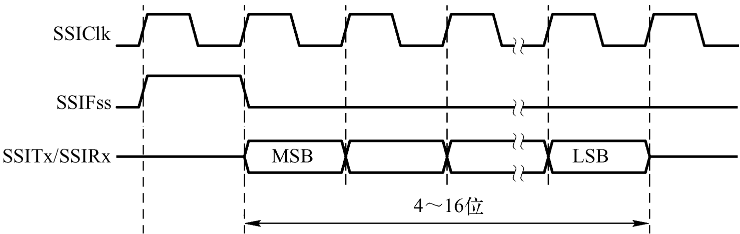


图 9-2　TI 同步串行帧格式的单次传输

在该模式中，当 SSI 模块处于空闲状态时，SSIClk 和 SSIFss 将被强制拉低，发送数据线 SSITx 被置为三态。在发送 FIFO 的底部入口含有数据时，SSIFss 将变为高电平并持续一个

SSIClk 周期。使待发送的值从发送 FIFO 中传输到发送逻辑的串行移位寄存器中。在 SSIClk 时钟的下一个上升沿，数据帧的 MSB 位将移位输出到从 SSITx 引脚上。同理，接收数据的 MSB 位也由片外串行从设备移位到 SSIRx 引脚上。

然后，SSI 和片外从设备在 SSIClk 的每一个下降沿时将数据位逐个移入到各自的串行移位器中。在锁存了 LSB 位后的第一个 SSIClk 上升沿接收到的数据从串行移位器传输到接收 FIFO 中。

TI 同步串行帧格式的连续传输如图 9-3 所示。

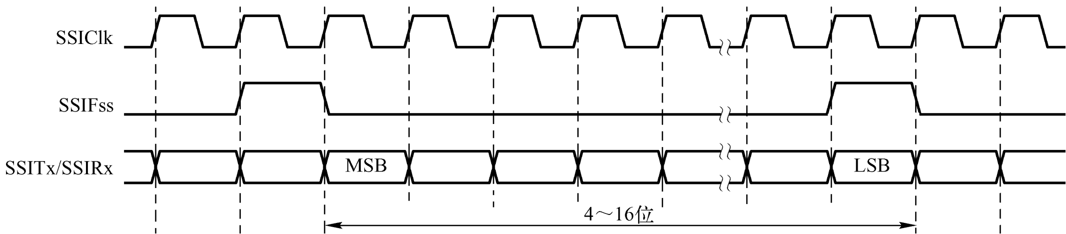


图 9-3　TI 同步串行帧格式的连续传输

（2）Freescale SPI 格式

Freescale SPI 接口为 4 线接口，其中 SSIFss 信号用于从设备选择。Freescale SPI 格式的主要特点是其 SSIClk 信号定义相当灵活，而非激活状态与相位可分别通过 SSISCR0 寄存器的 SPO/SPH 位进行设置。在操作 SSI 时需很好地理解下面两个位的意义：

① SPO 时钟极性位。在 SPO 时钟极性控制位清零时，将在 SSIClk 引脚上产生稳定的低电平。如果将 SPO 位置位，当未进行数据传输时，会在 SSIClk 引脚上产生稳定的高电平。

② SPH 相位控制位。SPH 相位控制位用于选择捕获数据的时钟沿，并允许数据改变状态。该位的状态对传输的首位影响最大，即是否在首次捕获数据之前忽略时钟的跳变。若 SPH 相位控制位清零，则在第一个时钟跳变沿捕获数据。若 SPH 位置位，则在第二个跳变沿捕获数据。

1）SPO =0 &SPH =0 时的 SPI 帧格式。在 SPO =0 与 SPH =0 时，SPI 帧格式的单次和连续传输信号时序如图 9-4 和图 9-5 所示。

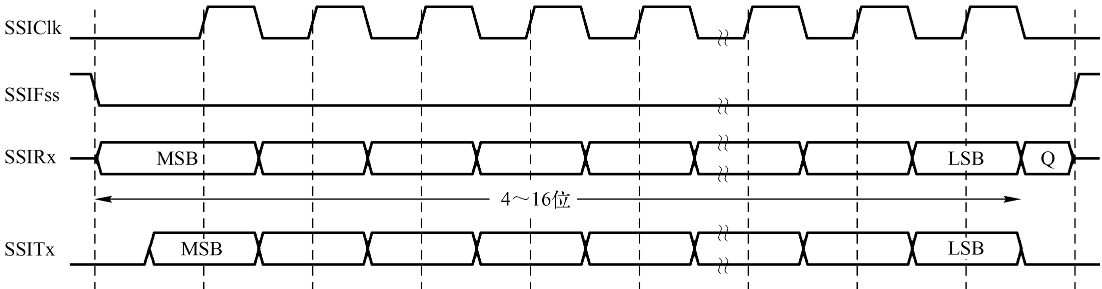


图 9-4　SPO =0 及 SPH =0 时 SPI 的帧格式（单次传输）

注：Q 为未定义。

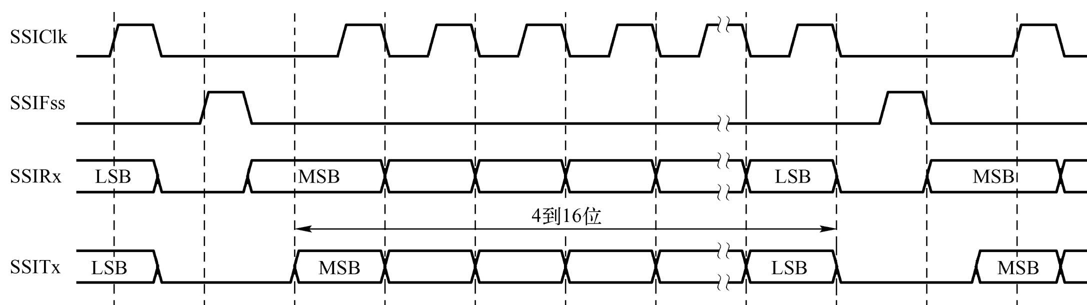


图 9-5　SPO = 0 及 SPH = 0 时 SPI 的帧格式（连续传输）

在该配置中，当 SSI 处于空闲时：

① SSIClk 被强制变为低电平。

② SSIFss 被强制变为高电平。

③ 发送数据线 SSITx 被仲裁强制变为低电平。

④ 当 SSI 配置为主机时，将使能 SSIClk 引脚。

⑤ 当 SSI 配置为从机时，将禁止 SSIClk 引脚。

若使能 SSI 模块，且发送 FIFO 中已填入有效数据，则在 SSIFss 主机信号被拉低时开始传输，使从机数据传输到主机的 SSIRx 输入线上，此时也将使能主机的 SSITx 引脚输出。

在半个 SSIClk 周期后，有效的主机数据将被传送到 SSITx 引脚。一旦主机和从机数据设置就绪，在另半个 SSIClk 时钟周期后，SSIClk 主机时钟引脚将被拉高。随后数据将在时钟信号的上升沿被捕获，而在时钟信号的下降沿进行传输。

对于单字传输，在数据字的所有位都被传输完后，SSIFss 线在捕获到最末一位后的一个时钟周期之后将返回到空闲的高电平状态。

而在背靠背的连续传输过程中，必须让 SSIFss 信号在相邻两次数据传输之间输出高脉冲，由于在 SPH 位为 0 时，从机选通引脚将锁定串行外设寄存器中的数据，不允许对其进行修改。所以主机必须在相邻两次数据传输之间拉高从机的 SSIFss 引脚，以使能串行外设数据的写入操作。在连续传输完成时，SSIFss 引脚在捕获到最末一位之后的一个时钟周期后将返回到空闲状态。

2）SPO = 0& SPH = 1 时 SPI 的帧格式。在 SPO = 0 与 SPH = 1 时，SPI 帧格式的传输信号时序如图 9-6 所示，该图涵盖了单次和连续两种传输情况。

在此配置中，当 SSI 处于空闲时：

① SSIClk 被强制变为低电平。

② SSIFss 被强制变为高电平。

③ 发送数据线 SSITx 被仲裁强制变为低电平。

④ 当 SSI 配置为主机时，将使能 SSIClk 引脚。

⑤ 当 SSI 配置为从机时，将禁止 SSIClk 引脚。

如果使能 SSI 并且在发送 FIFO 中包含有效数据，若 SSIFss 主机信号被驱动为低电平，将启动发送操作，同时也将使能主机 SSITx 引脚输出。当半个 SSIClk 时钟周期之后，主机和从机的数据都将在各自的发送线上就绪，并利用一个上升沿跳变来使能 SSIClk 时钟 。数

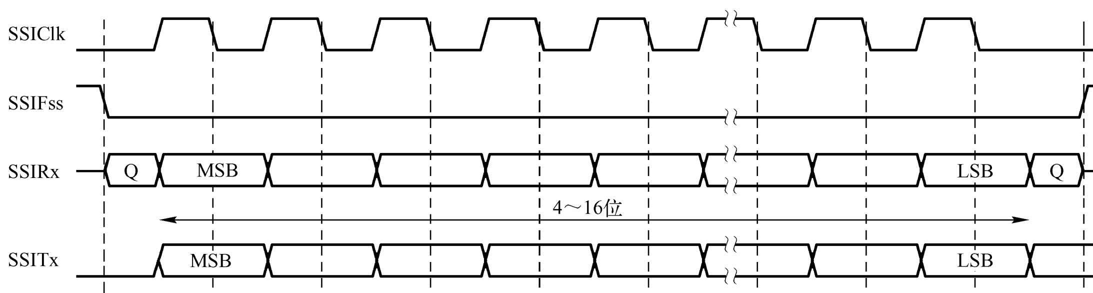


图 9-6　SPO = 0 和 SPH = 1 时 SPI 的帧格式

注：Q 为未定义。

据在 SSIClk 时钟信号的下降沿被捕获，而在时钟信号的上升沿进行传输。

对于单字传输，在所有位被传输完后，SSIFss 线将在捕获到最末一位（下降沿）后的一个 SSIClk 周期返回到空闲的高电平状态。

如果是连续传输，SSIFss 引脚将在连续数据字之间保持低电平，且连续传输的结束情况与单字传输的相同。

3）SPO = 1& SPH = 0 时 SPI 的帧格式。在 SPO = 1 与 SPH = 0 时，Freescale SPI 帧格式的单次和连续传输信号时序如图 9-7 和图 9-8 所示。

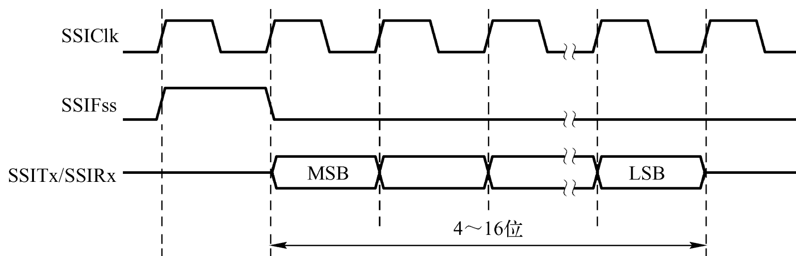


图 9-7　SPO = 1 及 SPH = 0 时的飞思卡尔 SPI 帧格式（单次传输）

注：Q 未定义。

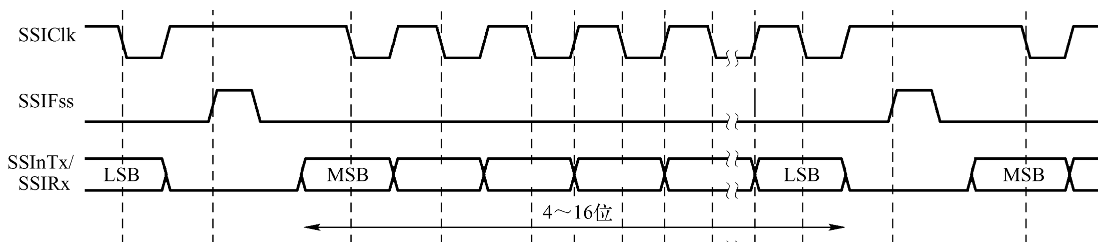


图 9-8　SPO = 1 及 SPH = 0 时的飞思卡尔 SPI 帧格式（连续传输）

在该格式配置下，当 SSI 处于空闲时：

① SSInClk 被强制拉高。

② SSInFss 被强制拉高。

③ 发送数据线 SSITx 被仲裁强制拉低。

④ 当 SSI 配置为主机时，将使能 SSIClk 引脚。

⑤ 当 SSI 配置为从机时，将禁止 SSIClk 引脚。

如果 SSI 模块已使能，并且发送 FIFO 中已包含有效数据，则在 SSInFss 主机信号被拉低时开始传输；使从机数据立即传输到主机的 SSInRx 线上，同时将使能主机设备的 SSInTx 输出端口。

在过去半个周期之后，有效的主机数据将传送到 SSInTx 线上。一旦主机和从机数据的设置就绪，再过去半个 SSInClk 周期，将使 SSInClk 主机时钟引脚变为低电平。即是说数据在每个 SSInClk 时钟的下降沿被捕获，而在上升沿进行传输。

对于单字传输，在数据字的所有位传输完成之后，当 SSInFss 引脚捕获到最末 1 位（下降沿）之后的 1 个 SSInClk 后将返回其空闲的高电平状态。

对于背靠背的连续传输，必须使 SSInFss 信号在相邻两次数据传输之间输出高脉冲，由于在 SPH 位清零时，从设备选通引脚将锁存串行外设寄存器中的数据，不允许对其修改。故主设备须在相邻两次数据传输之间拉高 SSInFss 引脚的电平，以使能串行外设数据的写入操作。当连续传输结束时，在 SSInFss 引脚捕获到最末 1 位后的 1 个 SSInClk 周期返回到空闲状态。

4）SPO＝1& SPH＝1 时 SPI 的帧格式。在 SPO＝1 与 SPH＝1 时，飞思卡尔 SPI 帧格式的传输信号时序如图 9-9 所示，该图涵盖了单次和连续两种传输情况。

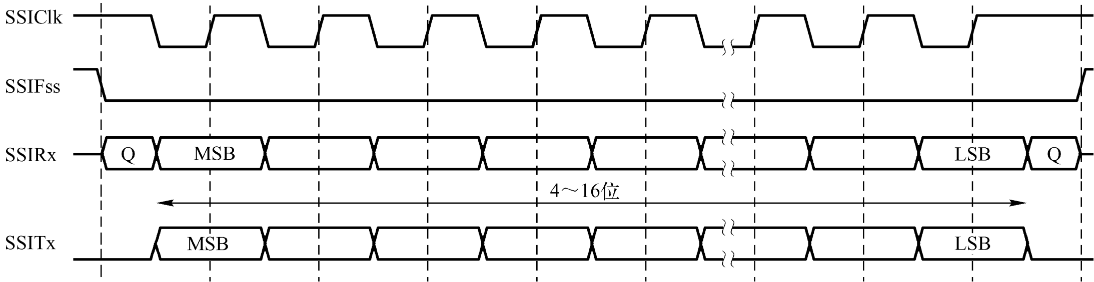


图 9-9　SPO＝1 及 SPH＝1 时的飞思卡尔 SPI 帧格式

注：Q 未定义。

在该格式配置下，当 SSI 处于空闲时：

① SSInClk 被强制拉高。

② SSInFss 被强制拉高。

③ 发送数据线 SSITx 被仲裁强制拉低。

④ 当 SSI 配置为主机时，将使能 SSIClk 引脚。

⑤ 当 SSI 配置为从机时，将禁止 SSIClk 引脚。

如果 SSI 已使能并且在发送 FIFO 中包含有效的数据，在 SSInFss 主机信号被驱动为低电平将启动发送操作。同时将使能主设备的 SSInTx 输出端口。再过半个 SSInClk 周期后，主设备和从设备的数据都将在各自的发送线上就绪，并使用下降沿的跳变将 SSInClk 使能。使数据在每个 SSInClk 时钟信号的上升沿被捕获，而在下降沿输出。

对于单字传输，当数据字的所有位传输完成后，在 SSInFss 引脚捕获到最末 1 位（上升

沿）后的 1 个 SSInClk 周期返回到空闲的高电平状态。

对于背靠背的连续传输，SSInFss 信号在连续的数据传输过程中将始终保持有效（低电平），直到捕获到最后 1 个字的最末 1 位（上升沿）之后将返回其空闲状态。

对于背靠背的连续，SSInFss 引脚将在连续的数据字之间保持低电平，且连续传输的结束情况与单字传输的相同。

5. DMA 操作

SSI 模块可与 μDMA 控制器接口具有相互独立的发送通道和接收通道。通过 SSIDMA 控制寄存器（SSIDMACTL）来使能 μDMA 操作。SSI 模块在接收 FIFO 或发送 FIFO 传输数据时可向接收或发送通道发出 μDMA 请求。对于接收通道，只要接收 FIFO 中存在数据，就会发出单次传输请求。若接收 FIFO 中的数据≥4 个，将会发出多个连续传输请求。对于发送通道，只要发送 FIFO 中存在一个空位，就会发出单次传输请求。如果发送 FIFO 中的空位≥4 个，将会发出多个连续传输请求。μDMA 控制器将根据 μDMA 通道的配置自动处理单次传输请求和连续传输请求。而在传输结束时 μDMA 控制器将会自动触发中断。

9.1.5 寄存器映射

表 9-2 列出了 SSI 模块中的寄存器。其中所列出的地址偏移量为寄存器相对于 SSI 的基址的十六进制增量：

- SSI0：0x4000.8000。
- SSI1：0x4000.9000。
- SSI2：0x4000.A000。
- SSI3：0x4000.B000。

注意：

① 配置这些寄存器之前必须先使能 SSI 模块时钟。

② 在 SSI 模块时钟使能之后必须等待 3 个系统时钟的延迟，方可访问 SSI 模块的寄存器。

③ 在对任何控制寄存器重新编程前，必须先行禁止 SSI 。

表 9-2　SSI 寄存器映射

偏移量	名　称	类　型	复　位	描　述
0x000	SSICR0	R/W	0x0000.0000	SSI 控制寄存器 0
0x004	SSICR1	R/W	0x0000.0000	SSI 控制寄存器 1
0x008	SSIDR	R/W	0x0000.0000	SSI 数据寄存器
0x00C	SSISR	RO	0x0000.0003	SSI 状态寄存器
0x010	SSICPSR	R/W	0x0000.0000	SSI 时钟预分频寄存器
0x014	SSIIM	R/W	0x0000.0000	SSI 中断屏蔽寄存器
0x018	SSIRIS	RO	0x0000.0008	SSI 原始中断状态寄存器
0x01C	SSIMIS	RO	0x0000.0000	SSI 屏蔽中断状态寄存器
0x020	SSIICR	WIC	0x0000.0000	SSI 中断清除寄存器

（续）

偏移量	名　称	类　型	复　位	描　述
0x024	SSIDMACTL	R/W	0x0000.0000	SSI DMA 控制寄存器
0xFC8	SSICC	R/W	0x0000.0000	SSI 时钟配置寄存器
0xFD0	SSIPeriphID4	RO	0x0000.0000	SSI 外设标识寄存器 4
0xFD4	SSIPeriphID5	RO	0x0000.0000	SSI 外设标识寄存器 5
0xFD8	SSIPeriphID6	RO	0x0000.0000	SSI 外设标识寄存器 6
0xFDC	SSIPeriphID7	RO	0x0000.0000	SSI 外设标识寄存器 7
0xFE0	SSIPeriphID0	RO	0x0000.0022	SSI 外设标识寄存器 0
0xFE4	SSIPeriphID1	RO	0x0000.0000	SSI 外设标识寄存器 1
0xFE8	SSIPeriphID2	RO	0x0000.0018	SSI 外设标识寄存器 2
0xFEC	SSIPeriphID3	RO	0x0000.0001	SSI 外设标识寄存器 3
0xFF0	SSIPCellID0	RO	0x0000.000D	SSI PrimeCell 标识寄存器 0
0xFF4	SSIPCellID1	RO	0x0000.00F0	SSI PrimeCell 标识寄存器 1
0xFF8	SSIPCellID2	RO	0x0000.0005	SSI PrimeCell 标识寄存器 2
0xFFC	SSIPCellID3	RO	0x0000.00B1	SSI PrimeCell 标识寄存器 3



9.2　SSI 固件库函数

SSI API 被分成若干函数组，一般包括以下几类：

① 模块配置。

② 数据处理。

③ 中断管理。

④ 函数更新。

（1）常用的 SSI 模块配置函数

- SSIConfigSetExpClk( )。
- SSIEnable( )。
- SSIDisable( )。
- SSIDMAEnable( )。
- SSIDMADisable( )。
- SSIClockSourceGet( )。
- SSIClockSourceSet( )。

（2）常用的数据处理函数

- SSIDataPut( )。
- SSIDataPutNonBlocking( )。
- SSIDataGet( )。
- SSIDataGetNonBlocking( )。

(3) 常用的中断管理函数

- SSIIntClear( )。
- SSIIntDisable( )。
- SSIIntEnable( )。
- SSIIntRegister( )。
- SSIIntStatus( )。
- SSIIntUnregister( )。

(4) 函数更新

老版本的 SSIConfig( )、SSIDataNonBlockingGet( )和 SSIDataNonBlockingPut( )API 函数被新版的 SSIConfigSetExpClk( )、SSIDataGetNonBlocking( )和 SSIDataPutNonBlocking( )API 函数所替代。宏已在 ssi. h 中提供老版本 API 到新版 API 的映射，允许已存在的应用程序链接和运行在新版 API 上。

详细的 SSI 固件库函数介绍请参考书后附录 G。



9. 3 例程

本小节将以 TI 的例程和网上有关数码管循环显示为例来简要介绍 SSI 固件库的使用方法。

1. 基于 SSI 的主机模式传输数据例程

1) 例程中用到 SSI0 和 UART0 的引脚图如图 9-10 所示。

引脚名称	引脚编号	引脚复用/赋值
SSI1C1k	30 61	PF2 PD0
SSI1Fss	31 62	PF3 PD1
SSI1Rx	28 63	PF0 PD2
SSI1Tx	29 64	PD1 PD3

图 9-10　SSI1 引脚分配图

2) ssi_master_transdata. c 介绍。

```
// ***********************************************************************
//文件名:ssi_master_transdata. c
//来源:TI 例程
//功能描述:
//! 该例将介绍如何配置 SSI1 作为主机 SPI 模式。并将 5 个十六进制数据通过 Tx 端口(即 PF1)
//!发送给板载红色 LED,以利用 LaunchPad 开发板现有的资源来观察 SSI 的数据传输,同时把要
//!发送的数据通过 UART 端口传送到 PuTTY 上显示
//! 此例程使用下列外设和 I / O 信号:
//! - SSI1 外设
//! - GPIOF 端口(SSI1 引脚)
```

```
//! - SSI1Clk - PF2
//! - SSI0Fss - PF3
//! - SSI0Tx - PF1
//!
//!下列配置 UART 信号仅为显示该例程的控制台消息：
//! - UART0 外设
//! - GPIO A 端口(UART0 引脚)
//! - UART0RX - PA0
//! - UART0TX - PA1

// ************************************************************************

#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/ssi.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
// ************************************************************************
//发送和接收的字节数
// ************************************************************************
#define NUM_SSI_DATA                5
uint8_t pui8DataTx[NUM_SSI_DATA] = {0xFF,0x00,0xF1,0x88,0x01};
// ************************************************************************
//此函数设置 UART0 的控制台来显示该例程运行时的信息
// ************************************************************************
void
InitConsole(void)
{
    //
    //使能用于 UART0 引脚功能的 GPIO 端口 A
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    //
    //将引脚复用端口 A0 和 A1 配置为 UART0 功能
    //
    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);

    //
    //使能 UART0 以便后续配置时钟
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    //
```

```
    //使用内部 16 MHz 振荡器作为 UART 时钟源
    //
    UARTClockSourceSet(UART0_BASE,UART_CLOCK_PIOSC);

    //
    //选择这些引脚的(UART)复用功能
    //
    GPIOPinTypeUART(GPIO_PORTA_BASE,GPIO_PIN_0 | GPIO_PIN_1);

    //
    //初始化 UART 的控制台 I/O
    //
    UARTStdioConfig(0,115200,16000000);
}

//*****************************************************************************
//将 SSI1 配置成飞思卡尔(SPI)的主机模式。这个例子将发送 5B 的数据来点亮板载红色 LED,以
及把待发送的数据通过 UART 在 PuTTY 上显示出来
//*****************************************************************************
int
main(void)
{
    uint32_t ui32Index;

    //
    //设置时钟直接从外部晶振运行
    //
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                SYSCTL_XTAL_16MHZ);

    //
    //设置串行控制台用于显示消息
    //
    InitConsole();

    //
    //在控制台上显示设置
    //
    UARTprintf("    通过 SSI 发送数据测试\n");
    UARTprintf("    模式: SPI\n");
    UARTprintf("    打印十六进制数据: \n\n");

    //
    //SSI1 外设使用时必须使能
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI1);

    //
```

```
    //使能 GPIOF
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

    //
    //将复用引脚 F1、F2、F3 配置成 SSI1 功能
    //
    GPIOPinConfigure(GPIO_PF2_SSI1CLK);
    GPIOPinConfigure(GPIO_PF3_SSI1FSS);
    GPIOPinConfigure(GPIO_PF1_SSI1TX);

    //
    //将 GPIO 设置为 SSI 引脚,且此函数也可通过这些引脚控制 SSI 硬件
    //
    GPIOPinTypeSSI(GPIO_PORTA_BASE,GPIO_PIN_1 | GPIO_PIN_3 | GPIO_PIN_2);

    //
    //设置：SSI、协议格式、工作模式、位速率及数据宽度
    //SSI1_BASE               -使用 SSI1
    //SysCtlClockGet()        -提供系统时钟
    //SSI_FRF_MOTO_MODE_0     -协议格式设置:SPO=0 和 SPH=0
    //SSI_MODE_MASTER         -工作模式设置:设置为主机模式
    //800                     -位速率为 800 Hz(便于能观察到红色 LED 闪烁)
    //16                      -数据宽度
    //
    SSIConfigSetExpClk(SSI0_BASE,SysCtlClockGet(),SSI_FRF_MOTO_MODE_0,
                       SSI_MODE_MASTER,800,16);

    //
    //使能 SSI1 模块
    //
    SSIEnable(SSI1_BASE);

    //发送 5B 数据
    //
  While(1)
  {
        for(ui32Index=0; ui32Index<NUM_SSI_DATA; ui32Index++)
        {
            //
            //打印正在传输的数据
            //
            UARTprintf("0x%x,",pui8DataTx[ui32Index]);

            //
            //使用“阻塞”put 函数发送的数据。在返回前,这个函数将一直等待到发
            //送 FIFO 中有空间为止。这可确保要发送的所有数据能放入到发送 FIFO 中
            //
            SSIDataPut(SSI0_BASE,pui32DataTx[ui32Index]);
```

```
                    //
                    //等待 SSI1 发送 FIFO 中的所有数据传输完成
                    //
                    while(SSIBusy(SSI1_BASE))
                    {
                    }
                }
            }
        }
```

3）建立 ssi_master_transdata 工程、编译工程、下载到 EK－TM4C123GXL 板中的测试结果，如图 9-11、图 9-12 所示。

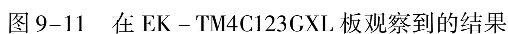


图 9-11　在 EK－TM4C123GXL 板观察到的结果



图 9-12　在 PuTTY 上显示正在发送的数据

从图 9-11 中可以看到，红色的 LED 不停的闪烁，该 LED 正好连接到 SSI1TX 引脚，说明有数据不断地传递到红色 LED，验证了程序实现了通过 SSI 的数据传输。有条件的读者可以在 SSI1TX 引脚处用示波器或逻辑分析仪观察传送的数据内容，并可通过修改传输速率来调节红色 LED 的闪烁频率。

为了使没有 EK - TM4C123GXL 板或示波器的读者也能观察到所传输数据的波形，下面将修改 ssi_master_transdata. c 程序，以便可以在 Proteus 虚拟硬件上运行。

4）修改过的 ssi_master_transdata. c 如下：

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_ssi.h"
#include "inc/hw_types.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"

#define NUM_SSI_DATA 5
const uint8_t pui8DataTx[NUM_SSI_DATA] =
{0xFF,0x00,0x0F,0x0F,  0x00};

int main(void)
{
    uint32_t ui32Index;

    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_6MHZ);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    GPIOPinConfigure(GPIO_PA2_SSI0CLK);
    GPIOPinConfigure(GPIO_PA3_SSI0FSS);
    GPIOPinConfigure(GPIO_PA5_SSI0TX);
    GPIOPinTypeSSI(GPIO_PORTA_BASE,GPIO_PIN_5|GPIO_PIN_3|GPIO_PIN_2);

    SSIConfigSetExpClk(SSI0_BASE,SysCtlClockGet(),SSI_FRF_MOTO_MODE_0,SSI_MODE_MAS-
TER,3000,16);
    SSIEnable(SSI0_BASE);

    while(1)
    {
        for(ui32Index=0; ui32Index<NUM_SSI_DATA; ui32Index++)
        {

            SSIDataPut(SSI0_BASE,pui8DataTx[ui32Index]);
```

```
                while(SSIBusy(SSI0_BASE))
                {
                }
            }

        }

    }
```

5）建立工程、编译、下载（略）。

6）搭建 SSI 数据传输程序测试电路，如图 9-13 所示。

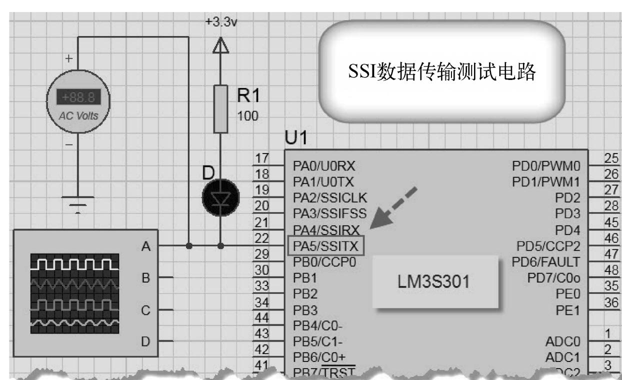


图 9-13　SSI 数据传输程序测试电路

7）SSI 数据传输程序的测试结果如图 9-14 所示。

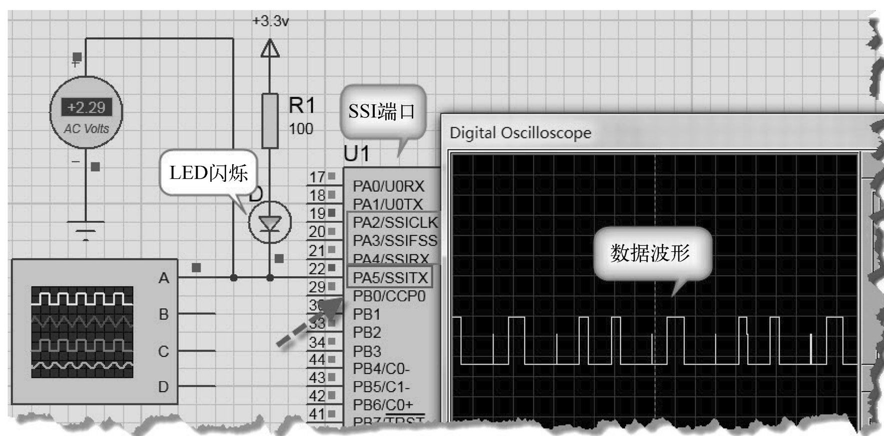


图 9-14　在 Proteus 中观察到的数据测试波形和 LED 灯的闪烁

注意：在做本实验时，需要安装 SW-DK-LM3S301-5961 软件包。该例程的工程将在配套资源中给出。

2. 基于 SSI 的数码管循环显示例程

（1）测试电路图

基于 SSI 的数码管显示程序的测试电路如图 9-15 所示，控制芯片可选 LM3S3xx 的任意

一款均可。

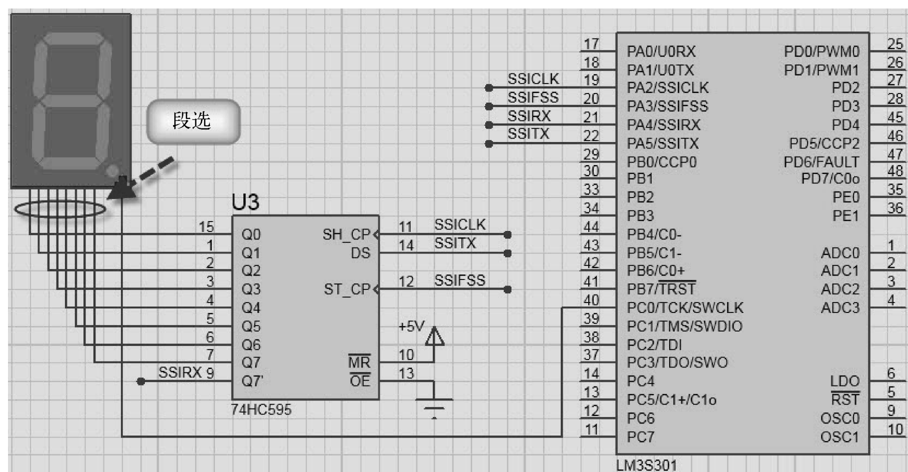


图 9-15　SSI 数码管程序的测试电路

（2）基于 SSI 的数码管显示程序

```
//文件名:ssi_digital_tube. c
//来源:由实验室根据 TI 例程及网络改编
//功能:利用 SSI 模块在数码管上动态显示 0 ~ A
//供无实际 M4 板卡的读者测试 SSI 模块代码之用

#include "inc/hw_ints. h"
#include "inc/hw_memmap. h"
#include "inc/hw_types. h"
#include "driverlib/debug. h"
#include "driverlib/gpio. h"
#include "driverlib/interrupt. h"
#include "driverlib/sysctl. h"
#include "driverlib/ssi. h"

//0 ~ A 的编码
unsigned char Digital_code[11] = {0xC0,0xF9,0xA4,0xB0,0x99,
                                  0x92,0x82,0xF8,0x80,0x90,0x88};//共阳

//{0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f,0x77//共阴

//使用 SSI 模块动态显示数字编码
int main(void)
{

    unsigned char i = 0;

      //设置 LDO 输出电压
    SysCtlLDOSet(SYSCTL_LDO_2_35V);
```

```
    //设置系统时钟
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_6MHZ);

    //使能片内 SSI 模块
    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI);

    //使能 GPIOA 端口
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    //使能 GPIOC 端口
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);

    //设置 GPIOC 端口上 的 PIN_0 引脚为输出
    GPIODirModeSet( SYSCTL_PERIPH_GPIOC,GPIO_PIN_0,GPIO_DIR_MODE_OUT);

    //设置 GPIOC 端口的 PIN_0 引脚具有 8mA 的驱动,并且弱上拉输出
    GPIOPadConfigSet( GPIO_PORTC_BASE,GPIO_PIN_0,
                      GPIO_STRENGTH_8MA,GPIO_PIN_TYPE_STD_WPU);

    //设置 SSI 为主机模式 0,8 位数据,波特率为 100000

    SSIConfigSetExpClk(SSI0_BASE,SysCtlClockGet(),SSI_FRF_MOTO_MODE_0,
                      SSI_MODE_MASTER,100000,8);

    //使能 SSI
    SSIEnable(SSI_BASE);

    //设置用于 SSI 的引脚
    GPIOPinTypeSSI(GPIO_PORTA_BASE,
                   GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5);

    //公共端信号
    GPIOPinWrite(GPIO_PORTC_BASE,GPIO_PIN_0,  0x01);

    //在七段数码管上循环显示 0 ~ A
    while(1)
    {
        for(i =0; i <12; i ++ )
        {
          SSIDataPut(SSI_BASE,Digital_code[i]);
          SysCtlDelay(120 * (SysCtlClockGet() / 3000));
        }
    }
}
```

(3) 创建 ssi_digital_tube 工程

读者可根据配套资源所给出的 SSI 数码管程序，按照前述方法自行完成。

（4）测试结果

SSI 数码管显示程序的测试结果如图 9-16 所示。

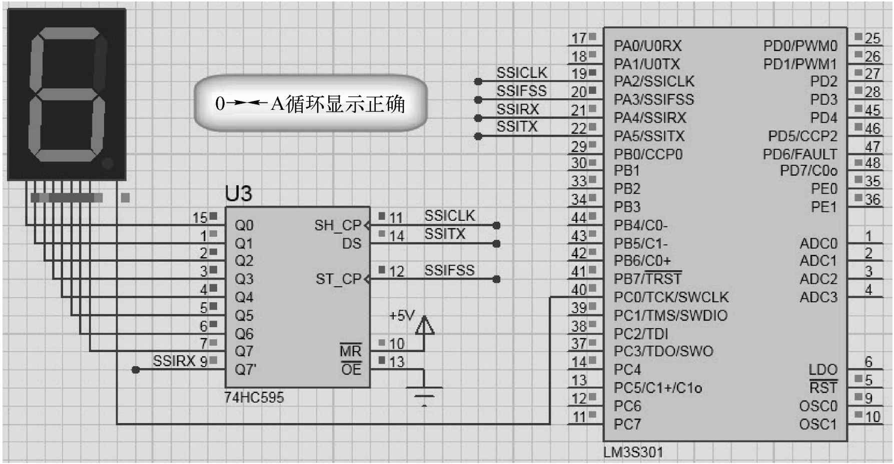


图 9-16　SSI 的数码管显示程序测试结果

说明： 本章例程在配套资源的第 9 章中给出。

第10章

内部存储器

内部存储器（Internal Memory）是微控系统中重要的器件之一，系统中所有程序的运行均是在存储器中进行的，因此存储器性能的高低对整个微控系统至关重要。TM4C123GH6PM 微控制器包含 32 KB 的位带（bit-banded）SRAM、内部 ROM、256 KB 闪存（Flash）、2 KB 的 EEPROM（为 M4 新增功能）。闪存控制器提供了一个友好的用户界面，使闪存编程成为简单的工作。闪存由 1 KB 大小独立可擦除的块组成，并以 2 KB 大小的块为单位应用存储器保护。EEPROM 模块提供了良好定义的寄存器接口，以支持随机读取或写入 EEPROM，以及支持轮询或顺序访问 EEPROM 的机制。密码模型允许应用程序锁定一个或多个 EEPROM 块来控制访问 16 字边界。

本章的主要内容：

- 内部存储器单元简介
- SRAM
- ROM
- 闪存
- EEPROM
- 内部存储器固件库函数（见附录 H）
- 例程



10.1 内部存储器单元

10.1.1 模块框图与控制逻辑

图 10-1 给出了 SRAM、ROM、闪存的模块框图和控制逻辑（其中点画线框表示位于系统控制模块中的寄存器）。

图 10-2 给出了 EEPROM 的模块框图和控制逻辑，这里 EEPROM 块连接到 AHB 总线。

10.1.2 功能简介

本小节将介绍 SRAM、ROM、闪存和 EEPROM 存储器的功能。

注意： μDMA 控制器可以传送和接收片上 SRAM 中的数据。由于闪存和 ROM 位于单独

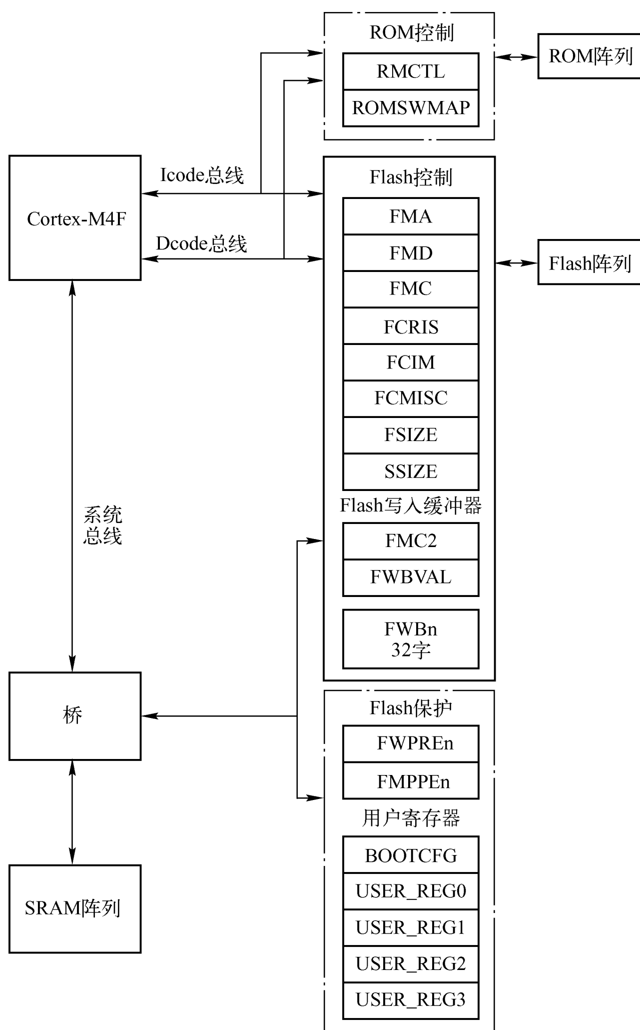


图 10-1　内部存储器模块框图与控制逻辑

的内部总线，μDMA 控制器不能从闪存或 ROM 来传送数据。

1. SRAM

TM4C123GH6PM 器件的内部 SRAM 位于器件存储器映射的地址为 0x2000.0000。为了减少耗费在读 - 修改 - 写中的操作时间，ARM 处理器提供了位带（bit - banding）技术。在已使能位带的处理器中，存储器映射中的某些区域（SRAM 和外设空间）能够使用地址别名，在单个原子操作中单独访问各个位。位带的基地址为 0x2200.0000。

计算位带别名的公式为

位带别名 = 位带基地址 + (字节偏移量 ×32) + (位编号 ×4)

例如，如果欲修改地址 0x2000.1000 的第 3 位，则位带别名的计算为

0x2200.0000 + (0x1000 ×32) + (3 ×4) = 0x2202.000C

由计算出的位带别名，对地址 0x2202.000C 执行的读/写指令可仅直接操作地址 0x2000.1000 字节的第 3 位即可。

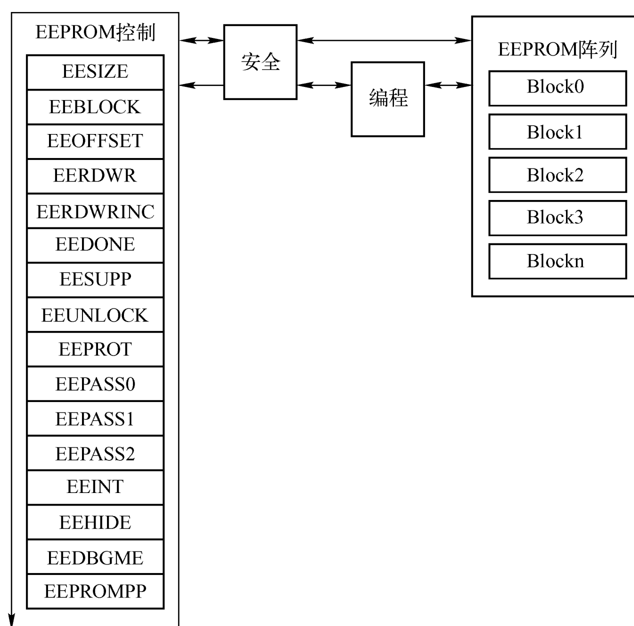


图 10-2　EEPROM 的模块框图和控制逻辑

2. ROM

TM4C123GH6PM 器件的内部 ROM 位于器件存储器映射地址 0x0100.0000 处。

ROM 包含以下组件：

① TivaWare 引导加载程序和向量表。

② TivaWare 外设驱动程序库（DriverLib）。

③ 高级加密标准（AES）密码表。

④ 循环冗余校验（CRC）错误检测功能。

引导加载程序是用来作为初始化程序的加载器（Flash 为空时），以及初始化应用程序固件的更新机制（通过回调引导加载程序）。应用程序可以调用 ROM 中的外设驱动函数库（API）来降低闪存操作的要求，并释放 Flash 存储器空间，以便实现其他用途（例如，应用程序中的附加功能）。高级加密标准（AES）是美国政府使用的一种公开定义的加密标准。循环冗余校验（CRC）为验证技术，用于判断一个数据块的内容是否与先前检验相同。

（1）引导加载程序概述

TivaWare 引导加载程序用来将代码下载到闪存设备，而不使用调试接口。当内核复位时，通过使能配置寄存器（BOOTCFG）中的 A ~ H 任意端口的 GPIO 信号，使用户有机会让内核直接执行 ROM 中的引导加载程序或闪存中的应用程序。

在复位时，将按以下顺序进行：

① 读 BOOTCFG 寄存器。若 EN 位被清零，将执行 ROM 中的引导加载程序。

② 在 ROM 的引导加载程序中，将指定的 GPIO 引脚状态与指定的极性进行比较。如果引脚状态符合指定的极性，则 ROM 将映射到地址 0x0000.0000，并继续执行 ROM 的引导加载程序。

③ 若 EN 位被置位或引脚状态不符合指定的极性，数据将从地址 0x0000.0004 读取，如

果在这个地址的数据是 0xFFFF. FFFF，则 ROM 将映射到地址 0x0000. 0000，并继续执行 ROM 的引导加载程序。

④ 如果在地址 0x0000. 0004 的数据值不是 0xFFFF. FFFF，堆栈指针（SP）将从闪存的地址 0x0000. 0000 处加载，而程序计数器（PC）则从地址 0x0000. 0004 处装载。然后用户应用程序开始执行。

引导加载程序使用一个简单的包接口与设备进行同步通信。引导加载程序的速度由内部振荡器（PIOSC）的频率决定，因为它不会使能 PLL。其可用于以下的串行接口：

① UART0。

② SSI0。

③ I2C0。

④ USB。

这里 UART0、SSI0、I2C0 接口使用相同的数据格式和通信协议。

注意：引导装载程序的 Flash 存储器驻留版本也支持 CAN 。

（2）TivaWare 的外设驱动库

TivaWare 外设驱动库包含 driverlib/rom. h 文件，它帮助调用 ROM 中的外设驱动库函数。每个函数的详细描述请参考“TM4C123GH6PM ROM User’s Guide”。更多有关调用 ROM 函数和使用 driverlib/rom. h 的详细信息，请参考“SW – TM4C – DRL – UG – 2. 0. 1. 11577”中的“使用 ROM”。当使用不同的 Tiva C 系列设备时，在 ROM 中可能有不同的 DriverLib 函数子集，提供的 DriverLib/ rom_map. h 头文件可增强其移植性。

其他 API 函数可用于图形和 USB 功能，但不会预装载到 ROM 中。TivaWare 图形库提供了一组用于基本图形设置和小工具设置，为基于 Tiva C 系列微控制器具有图形显示功能的开发板创建图形用户界面，详细的信息请参考 TI 官方文献：SW – TM4C – GRL – UG – 2. 0. 1. 11577。TivaWare USB 库是一组数据类型和函数，可用于在装有 Tiva C 系列微处理器的开发板上创建 USB 设备、主机或 OTG 应用，详细的信息请参考 TI 官方文献：SW – TM4C – USBL – UG – 2. 0. 1. 11577。

（3）高级加密标准（AES）密码表

AES 是一种强大的加密方法，它在硬件和软件方面都很快速且便于使用，仅需很少的存储空间即可。对于可预先安排密钥的应用程序，采用 AES 是比较理想的。ROM 中提供了执行 XySSL AES 所需要的四个数据表：

① 正向的 S – box 替换表。

② 反向的 S – box 替换表。

③ 正向的多项式表。

④ 反向的多项式表。

（4）循环冗余检验（CRC）错误检测

CRC 技术可用来确认接收的信息是否正确，用于确认解压后的数据与验证 Flash 存储器的内容是否更改，以及其他数据需要被确认的情况。

3. 闪存（Flash）存储器

（1）闪存存储器概述

闪存 API 提供了一组用于处理片上闪存的函数。这些函数用来编程和擦除闪存、配置闪

存保护，以及处理闪存中断。

当系统的时钟速度为 40 MHz 或以下时，Flash 存储器是单周期读取的。闪存是一系列大小为 1 KB 可被单独擦除块的集合。擦除一个块会使该块的内容重置为全 1。这些块可配对成一组可被单独保护的 2 KB 块的集合，且可标记为只读或只执行，并提供不同级别的代码保护。只读块不能被擦除或者编程，以防止这些块的内容被修改。只执行块同样不能被擦除或者编程，并且只能由控制器指令机制读取，以防止这些块的内容被控制器或调试器读取。

（2）预取指缓冲器

闪存控制器有一个预取指缓冲器，当 CPU 频率大于 40 MHz 时，它会自动开启。在该模式下，闪存将在二分之一系统时钟下工作。每个时钟周期，预取指缓冲器将获取两个 32 位字，这样在代码线性执行时取指不会处于等待状态。取指缓冲器包含一个分支推断机制，可辨认出分支，从而避免因读取下一对字而增加额外的等待状态。致使短循环分支常常保留在缓冲器中。因此，一些分支可无需等待便可执行，而其他分支需要一个单独的等待状态。

（3）闪存存储器保护

用户可以在 四对 32 位宽存储器中使用两种闪存存储器保护形式，以 2 KB 闪存存储器块为单位。FMPPEn 和 FMPREn 寄存器的各个位控制各种保护形式的策略（每个块控制一种策略）。

① 闪存存储器保护编程使能寄存器（FMPPEn）：若某个位被置位，就可对相应的块进行编程（写入）或擦除。如果某个位被清零，则不能修改相应的块。

② 闪存存储器保护读取使能寄存器（FMPREn）：若某个位被置位，则软件或调试器就可对相应的块执行或读取。如果某个位被清零，则相应的块只能被执行，块的内容禁止被作为数据读取。

闪存存储器的保护模式见表 10-1。

表 10-1 闪存存储器的保护模式

FMPPEn	FMPREn	保　　护
0	0	只执行保护。模块只能被执行，不能被写入或擦除。该模式用来保护代码
1	0	模块既可被写入，也可被擦除或执行，但不能被读取。该模式很少使用
0	1	只读保护。模块既可被读取，也可被执行，但不能被写入或擦除。该模式用于锁定模块防止对其进行修改，但允许对其执行任意的读或执行访问
1	1	无保护。在该模式下，可对模块进行写入、擦除、执行或读取

（4）闪存编程

Tiva C 设备为闪存编程提供了一个友好的用户接口。所有的擦除/编程操作都通过 3 个寄存器来处理：闪存地址（FMA）、闪存数据（FMD）和闪存控制（FMC）。

注意：如果微控制器的调试功能未被激活而处于“锁死”状态，必须执行一段恢复序列来激活调试模块。请参考 TI 官方技术文档 Tiva TM4C123GH6PM Microcontroller DATA SHEET 中的 201 页有关恢复锁定微控制器的内容。

在闪存操作（写、页擦除或整体擦除）期间，禁止访问闪存。作为结果，指令和按字取指将延迟到闪存操作完成。如果在闪存操作期间需要执行指令，则代码必须放置在 SRAM

上，并在 SRAM 上执行。

（5）中断

闪存控制器在检测到下列状态时会产生中断：

① 编程中断。当编程或擦除动作完成时发出的信号。

② 访问中断。当对受相应 FMPPEn 位保护的 2 KB 块存储器尝试编程或擦除操作时发出的信号。

能触发控制器级中断的事件在闪存控制器屏蔽中断状态寄存器（FCMIS）中定义，将相应的 MASK 位置位可使能该中断。若未使用中断，则原始的中断状态将在闪存控制器原始中断状态寄存器（FCRIS）中可见。

对闪存控制器屏蔽中断的状态和清除寄存器（FCMISC）中相应的位写 1 将清除相应的中断（适用于 FCMIS 和 FCRIS 寄存器）。

（6）基本编程/擦除操作

1）对 1 个 32 位字进行编程的步骤：

① 将源数据写入 FMD 寄存器。

② 将目标地址写入 FMA 寄存器。

③ 将闪存写密钥和 WRITE 位的值写入到 FMC 寄存器中。根据 BOOTCFG 寄存器中 KEY 位的值，在进行闪存的写操作时，必须将值 0xA442 或 0x71D5 写入到 WRKEY 字段。

④ 查询 FMC 寄存器，直到 WRITE 位清零。

2）执行 1 KB 页擦除的步骤：

① 将页地址写入到 FMA 寄存器中。

② 将闪存写密钥和 ERASE 位的值写入到 FMC 寄存器中。根据 BOOTCFG 寄存器中 KEY 位的值，在进行闪存的写操作时，必须将值 0xA442 或 0x71D5 写入到 WRKEY 字段。

③ 查询 FMC 寄存器直到 ERASE 位清零，或者用 FCIM 寄存器中的 PMASK 位使能编程中断。

2）执行整体闪存擦除的步骤：

① 将闪存写密钥和 MERASE 位的值写入到 FMC 寄存器中。根据 BOOTCFG 寄存器中 KEY 位的值，在进行闪存的写操作时，必须将值 0xA442 或 0x71D5 写入到 WRKEY 字段。

② 查询 FMC 寄存器直到 MERASE 位清零，或者用 FCIM 寄存器中的 PMASK 位来使能编程中断。

（7）32 个字的闪存写缓存器

32 字的写缓冲器提供了同时编程 2 个 32 位字的能力，可加快闪存写访问的速度，可在处理 16 字的同时对 32 字进行编程，需缓存的数据被写入到写缓冲器寄存器（FWBn）中。

这些寄存器在闪存中是以 32 字对齐的，则 FWB0 寄存器对应 FMA 中的地址（FMA 的［6:0］位全为 0）。FWB1 寄存器对应 FMA + 0x4 中的地址，以此类推。仅在上次缓存的闪存进行写入操作后，更新过的 FWBn 寄存器方能被写入。闪存写缓冲器有效寄存器（FWB-VAL）显示了从上次缓存闪存写操作后，哪些寄存器已被写入。该寄存器包含的位对应 32 个 FWBn 寄存器，其中 FWBVAL 的第［n］位对应 FWBn。若 FWBVAL 寄存器中的某位被置位，则表明相应的 FWBn 寄存器已被更新过了。

采用单独被缓冲的闪存写入操作来编程 32 个字的步骤：

① 将源数据写入到 FWBn 寄存器中。

② 将目标地址写入到 FMA 寄存器中。该地址必须为一个 32 字对齐的地址（即 FMA 的［6:0］位为全 0）。

③ 将闪存写入密钥和 WRBUF 位写入到 FMC2 寄存器中（即写入 0xA442.0001）。

④ 查询 FMC2 寄存器，直到 WRBUF 位清零，或者等待发出 PMIS 中断信号。

（8）非易失性寄存器编程

下面将讨论如何更新闪存自身的寄存器。这些寄存器存在于主闪存阵列之外的单独空间中，它不受擦除或整体擦除的影响。写操作可将这些寄存器中的位由 1 变 0。除了上电复位会将寄存器的内容复位为 0xFFFF.FFFF 外，其他复位都不会影响寄存器的值。在上电序列后，可用 FMC 寄存器中的 COMT 位来提交确认寄存器的值，之后寄存器的值将变为非易失而保持下去。在寄存器的内容被提交后，能恢复出厂默认值的唯一方法就是执行“恢复一个“锁定的”微控制器”中描述的方法（见 TI 文档 Datasheet－TM4C123GH6PM 的 201 页）。

该驱动包含在 driverlib/flash.c 中，driverlib/flash.h 包含 API 的定义。

4. EEPROM

（1）EEPROM 模块的特点

TM4C123GH6PM 微控制器包括 1 个 EEPROM 模块，具有以下特点：

① 可访问 512 个 32 位字的 2 KB 内存。

② 32 个 16 字的块（64 B）。

③ 每个 2 页面块可进行 500 K ～ 15 M 次写操作。

④ 每个块的访问保护。

⑤ 整个外设的锁定保护选项与每个块的锁定保护相同，使用 32～96 位的解锁码。

⑥ 支持写入完成后产生中断，以避免轮询操作。

⑦ 内置平衡擦写。

（2）功能简述

EEPROM 模块提供了一个定义良好的寄存器接口来支持随机读取或写入 EEPROM，以及支持轮询或顺序访问 EEPROM 的机制。

一种保护机制允许锁定 EEPROM 块，以防止在某些情况下不必要的写入或者读取操作。密码模型允许应用程序锁定一个或多个 EEPROM 块来控制访问 16 字边界。

1）锁定和密码。可对 EEPROM 在模块级和块级进行锁定。由 EEPROM 的密码寄存器（EEPASSn）中的密码控制，该密码为一个 32 ～96 位之间的值，但不可全是 1。其中，块 0 为主块，不仅用它的密码可对控制寄存器与其他的块进行保护，而且也可用块密码对每个块进行进一步的保护。

若块 0 设有密码，则在复位时整个模块都将被锁定。其锁定规则：在块 0 解锁以后，将使块 1～块 31 可访问，而块 0 会继续受自己的保护位保护。任何块（包括块 0）设置密码后，都可以根据是否锁定控制块来决定能否被访问。

复位时所有密码保护的块均会被锁定。欲解锁这些块，就必须将正确的密码写入 EEPROM 解锁寄存器（EEUNLOCK）中。写入密码时，应通过 EEPASSn 寄存器将其写入 1 ～ 3 遍来组成 32 位、64 位或 96 位的注册密码。在 EEUNLOCK 寄存器中写入 0xFFFF.FFFF 可以

重新锁定块或模块（0xFFFF. FFFF 为无效密码）。

2）保护与访问控制。保护位为每个块提供了单独的读写控制，可实现的保护模式如下：

① 没有密码：任何时间都可进行读/写操作，在无密码时，该模式为默认。

② 没有密码：可以读取，但不能写入。

③ 有密码：可读取，只有在密码解锁后方可写入，在有密码时，该模式为默认。

④ 有密码：在锁定时只能读取，或只能写入。

⑤ 有密码：在锁定时只读，但不能写入。

另外，访问保护还可以通过处理器模式实现。这种配置允许仅管理员访问，或管理员与用户访问（默认模式）。其中，仅管理员访问模式还可以阻止 μDMA 与调试器的访问。

同时，主块还可以用来控制保护机制本身的访问保护。若块 0 的访问控制是仅管理员访问模式，则整个模块都可以用管理员模式访问。并且块 0 的保护级别为整个 EEPROM 设置了最低的保护级别。

3）中断控制。EEPROM 模块允许在写入完成之后发出中断，而无需轮询。该中断可用于驱动应用程序的 ISR，使 ISR 可写入更多的字或验证操作的完成。无论是由于错误还是成功编程或擦除操作所引起，该中断机制均可在 EEDONE 寄存器工作或停止时随时使用。EEPROM 中断用闪存中断向量给内核发信号，软件可通过检测闪存控制器屏蔽中断状态和清除寄存器（FCMISC）的第 2 位来确定中断源是否为 EEPROM。

4）工作原理。EEPROM 采用 EEPROM 类型单元的传统闪存块模型操作，但使用扇区擦除。另外，若需要，在复制页中的字时，允许 500k + 个擦除周期，即每个字均有一个最新版本。这样，写入操作就在新的位置创建了一个新版本的字，使其原先的值被抛弃。

每个扇区包含两个块，每个块包含一个活动副本和六个冗余副本的位置。将密码、保护位和控制数据都保存在页中。当页中保存最新字的空间不足时，就会启用一个复制缓冲区，使其复制每个块中的最新字，并将原来的页擦除，最终将复制缓冲区的内容复制回页中。这种机制可确保掉电时的数据不丢失，即使在操作期间。EEPROM 机制也会跟踪所有的状态信息，提供全面的安全性和保护。尽管几率很小，但在某些环境下还是可能发生错误，比如，编程过程中电压过低等。

EEPROM 详细的功能描述请参考 Datasheet – TM4C123GH6PM 的数据手册（见配套资源）。



10.2 闪存固件库函数

闪存的 API 被分成三个函数组：

① 闪存编程。

② 闪存保护。

③ 中断处理。

常用的一些 API 函数列举如下，较详细功能介绍请见附录 H 的闪存部分。

（1）闪存编程

- FlashErase( )。
- FlashProgram( )。
- FlashUsecGet( )。
- FlashUsecSet( )。

（2）闪存保护

- FlashProtectGet( )。
- FlashProtectSet( )。
- FlashProtectSave( )。

（3）中断处理

- FlashIntRegister( )。
- FlashIntUnregister( )。
- FlashIntEnable( )。
- FlashIntDisable( )。
- FlashIntGetStatus( )。
- FlashIntClear( )。



10.3 使用 ROM

让部分外设驱动库保存到片上 ROM 中，从而可使替换出的闪存空间用于应用程序。此外，引导装载程序也可存放在 ROM 中，使之被应用程序调用来启动固件库的更新。

本小节介绍的内容如下：

- 直接 ROM 调用。
- 映射 ROM 调用。
- 固件升级。

较详细功能介绍请见附录 H 的使用 ROM 部分。

10.3.1 直接 ROM 调用

（1）步骤

① 将待运行应用程序的器件必须使用预处理符号定义，该过程可在源代码或在编译应用程序的工程中完成。如果代码是在工程之间共享，后者将更加灵活。

② 头文件 driverlib/rom.h 中将包含要调用 ROM 的源代码。

③ 调用 ROM 外设驱动库函数的版本。例如，如果在 ROM 中调用 GPIODirModeSet( )可用 ROM_GPIODirModeSet( )替代。

（2）定义（define）

用来选择要使用的设备，因为在 ROM 中的一组可用的函数集合必须有相应定制的编译选项；在运行时的检查不会提供任何闪存保存，因为无论是 ROM 调用还是闪存版本的 API 都将在闪存映像中。

通过使用 ROM_Function( )，ROM 将被显示的调用。如果讨论的函数访问不在 ROM 中，

将产生一个编译错误。

针对特定设备在 ROM 中提供的 API 的详细信息，请参阅 TivaWare ROM 用户指南（spmu349）。

（3）样例

下面是一个在 ROM 中调用函数的例子，用#define 在源代码中定义讨论的设备以替代工程中的文件：

```
#define TARGET_IS_DUSTDEVIL_RA0
#include "driverlib/rom.h"
#include "driverlib/systick.h"
int
main(void)
{
ROM_SysTickPeriodSet(0x1000);
ROM_SysTickEnable();
//…
}
```

10.3.2 映射 ROM 调用

在工程之间共享代码时，且一些工程在有 ROM 的设备上运行，而另一些工程却在没有 ROM 的设备上运行，方便的做法是使代码能自动调用 ROM 或者闪存版本的 API 函数，而无需在代码中使用#ifdef－s。rom_map.h 提供了一个访问 ROM 的自动映射特性。类似 rom.h 提供的 ROM_Function()API，rom.h 也提供一组 MAP_Function()API 函数。如果函数在 ROM 中，MAP_Function()只需简单地调用 ROM_Function()即可，否则将调用 Function()。

（1）步骤

执行映射 ROM 调用的步骤如下：

① 遵循以上包含与使用 driverlib/rom.h 的步骤。

② 包含 driverlib/rom_map.h。

③ 继续上述例程，在源编码中调用 MAP_GPIODirModeSet()。

与直接调用 ROM 的方法类似，在编译时需作出调用 ROM 还是 Flash 版本的选择。仅通过 ROM 映射功能所提供的 API 在 ROM 中可用，即其不是在所有的外设 API 函数中均可使用。

（2）样例

下面是在共享代码中调用函数的例子，其中讨论的设备在工程文件中定义：

```
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/systick.h"
void
SetupSysTick(void)
{
MAP_SysTickPeriodSet(0x1000);
Map_SysTickEnable();
```

```
}
```

在编译不含 ROM 设备时，这个例子等效于：

```
#include "driverlib/systick.h"
void
SetupSysTick(void)
{
SysTickPeriodSet(0x1000);
SysTickEnable();
}
```

在编译包含 ROM 设备时，这个例子等效于：

```
#include "driverlib/rom.h"
#include "driverlib/systick.h"
void
SetupSysTick(void)
{
ROM_SysTickPeriodSet(0x1000);
ROM_SysTickEnable();
}
```

10.3.3 ROM 固件更新

ROM 固件更新见表 10-2 ~ 表 10-4。

表 10-2 ROM_UpdateI2C

功　　能	从 I2C0 接口启动更新
函数原型	void ROM_UpdateI2C(void)
描述	通过 I2C0 接口调用该函数开始一个固件的更新。此函数假设已经配置好了 I2C0 接口，并且目前正在运行。I2C0 从设备用于数据传输，I2C0 主设备用于监控总线忙状态
返回参数	无

表 10-3 ROM_UpdateSSI

功　　能	从 SSI0 接口启动更新
函数原型	void ROM_UpdateSSI(void)
描述	通过 SSI0 接口调用该函数开始一个固件的更新。此函数假设已经配置好了 SSI0 接口，并且目前正在运行
返回参数	无

表 10-4 ROM_UpdateUART

功　　能	从 UART0 接口启动更新
函数原型	void ROM_UpdateUART(void)
描述	通过 UART0 接口调用此函数开始一个固件的更新。此函数假设已经配置好了 UART0 接口，并且目前正在运行
返回参数	无



10.4 EEPROM 固件库函数

（1）EEPROM 概述

EEPROM API 提供了一组易于使用片上非易失数据存储器“EEPROM”的函数。该函数用于编程和擦除 EEPROM、配置 EEPROM 保护、处理 EEPROM 中断。

可对 EEPROM 以逐字“逻辑与”的方式进行编程，与闪存不同，在 EEPROM 中写一个新值之前不需要应用程序显式地擦除一个字或页。

当尝试一个无效访问时（如读取受保护的块），EEPROM 控制器将会发出一个中断。这个中断可被用来验证编程操作，以防止自动忽略无效访问，从而隐藏潜在的 bug。并且当一个擦除或编程操作完成时，将会产生中断。

可用的 EEPROM 大小和它包含的块数目在 Tiva 家族的不同成员之间是有差异的。用函数 EEPROMSizeGet( ) 和 EEPROMBlockCountGet( ) 在运行时确定这些信息。

可在设备和块级别配置密码保护用于读取和写入访问控制来支持数据保护。此外，块可被配置为仅当 CPU 运行在管理员模式时才允许访问。第二个保护机制，允许一个或多个 EEPROM 块在下一次系统复位前，完全不能使用软件访问。

此驱动程序包含在 driverlib/eeprom.c 中，另外 driverlib/eeprom.h 包含应用程序使用的 API 定义。

（2）EEPROM 固件库函数

EEPROM 的 API 被分成 4 个函数组：

① 读 EEPROM。

② 编程 EEPROM。

③ 保护 EEPROM。

④ 中断处理。

常用的一些 API 函数列举如下，较详细功能介绍请见附录 H 的 EEPROM 部分

（1） 读 EEPROM

- EEPROMRead( )。

（2） 编程 EEPROM

- EEPROMMassErase( )。
- EEPROMProgram( )。
- EEPROMProgramNonBlocking( )。

（3）保护 EEPROM

- EEPROMBlockProtectGet( )。
- EEPROMBlockProtectSet( )。
- EEPROMBlockPasswordSet( )。
- EEPROMBlockLock( )。
- VEEPROMBlockUnlock( )。
- EEPROMBlockHide( )。

(4) 中断处理

- EEPROMIntEnable()。
- EEPROMIntDisable()。
- EEPROMIntStatus()。
- EEPROMIntClear()。



10.5 例程

10.5.1 写闪存例程

本小节将以 TI 的例程为例来介绍 Flash 闪存的固件库使用方法。

1) 闪存程序。

```
*****************************************************************************
//!文件名:write_flash.c
//!来源:  根据 TI 例程改编
//!功能描述:向闪存写入数据并将写入的数据读出,显示在 PuTTY"串口助手上"
*****************************************************************************

#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/flash.h"

//****************************************************************************
//!初始化控制台函数
//****************************************************************************
void
InitConsole(void)
{
    //
    //使能 GPIOA 的 UART0 引脚
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    //
    //配置复用引脚 A0 和 A1 为 UART0 功能
    //
    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);
```

```
    //
    //使能 UART0 以便可以配置时钟
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    //
    //使用内部 16 MHz 振荡器作为 UART 时钟源
    //
    UARTClockSourceSet(UART0_BASE,UART_CLOCK_PIOSC);

    //
    //选择作为 UART 功能的引脚
    //
    GPIOPinTypeUART(GPIO_PORTA_BASE,GPIO_PIN_0 | GPIO_PIN_1);

    //
    //初始化 UART 的控制台 I/O
    //
    UARTStdioConfig(0,115200,16000000);
}

int main(void)
{
    //定义写入和读回数据的类型
    uint32_tpui32Data[2];
    uint32_tpui32Read[2];

    //初始化待写入的数据
    pui32Data[0] = 0x123456ab;
    pui32Data[1] = 0x7891011c;

    //配置系统时钟频率为 40 MHz
    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|
                SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

****************************************************************************
//! 向闪存中写入数据
****************************************************************************
    //擦除闪存中的 1 个块
    FlashErase(0x10000);

    //在 PuTTY 中显示待写入闪存中的数据
    InitConsole();
    UARTprintf("例程中待写入和从闪存中读回的数据:\n\n");
    UARTprintf("显示写入的数据:");
    UARTprintf(" %02x",pui32Data[0]);
    UARTprintf(" %02x",pui32Data[1]);
    UARTprintf("\n");
```

```
        //在新擦除的块中写入数据
        FlashProgram(pui32Data,0x10000,sizeof(pui32Data));

        //读回已写入到闪存中的数据
        pui32Read[0] = ( * ((volatile unsigned long * )0x00010000));
        pui32Read[1] = ( * ((volatile unsigned long * )0x00010004));

        //显示读回已写入闪存的数据
        UARTprintf("显示读出的数据:");
        UARTprintf(" %02x",pui32Read[0]);
        UARTprintf(" %02x",pui32Read[1]);
        UARTprintf("\n");

        //循环等待
        while(1)
        {
        }
    }
```

2） 创建 write_flash 工程。

3） 编译 write_flash 工程生成可执行的 . out 格式文件。

4） 在 LaunchPad（EK－TM4C123GXL）板上对程序进行调试。

① 将编译生成的 . Out 文件导入 LaunchPad 板中。

② 在 22 行处设置一个断点，然后单步执行程序，当程序指针位于 25 行时的运行结果如图 10-3 所示。

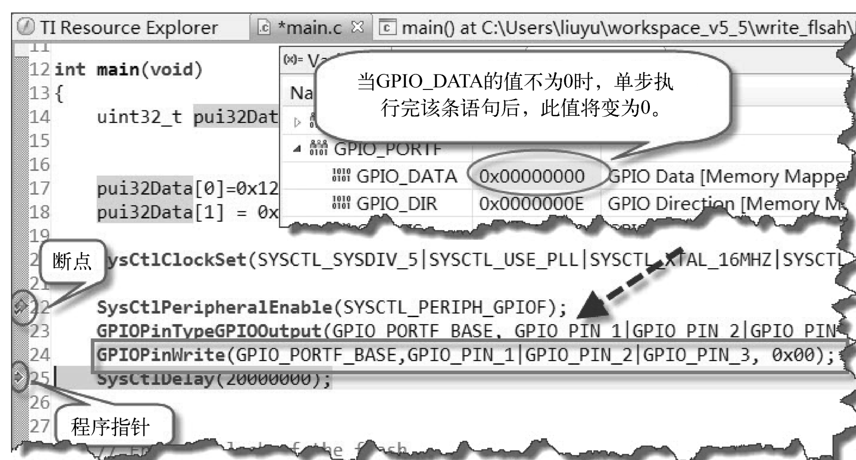


图 10-3　清零 GPIO_DATA 寄存器

从图 10-3 中可以看到，如果 GPIO_DATA 寄存器的值不为 0，当程序执行完 GPIOPin-Write()后，该寄存器将被清零。因此，观察到的结果都符合程序所表达的含义。

③ 在 29 行 FlashErase(0x10000)处设置一个断点，当单步执行完该语句后的结果如图 10-4 所示。



图 10-4　擦除地址 0x10000 处的数据

从图 10-4 可以看到，当执行完该条语句后的确擦除了地址为 0x10000 处的数据。然后继续单步执行，当执行完 FlashProgram( )语句后的结果如图 10-5 所示。

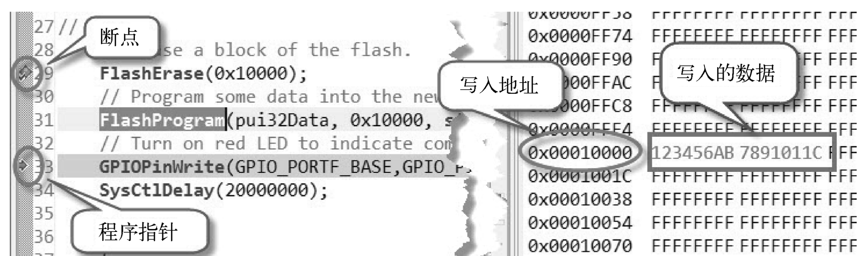


图 10-5　向地址 0x10000 写入的数据

从图 10-5 可以看到，写入的数据和从闪存中读出的数据完全相同，验证了程序功能正确。

5）在 LaunchPad（EK－TM4C123GXL）板上对程序进行测试。将编译生成的 . out 文件下载到 LaunchPad 板中，让程序全速在 LaunchPad 板中运行。从地址 0x10000 和 0x10004 读出的闪存数据和程序中待写入的数据相同（见图 10-5）。这时打开 PuTTY“串口助手”，其测试结果如图 10-6 所示。



图 10-6　待写入和从闪存中读回数据的测试结果

从图 10-6 中可以看到写入到闪存中的数据和读出的数据完全相同，验证了 write_flash. c 程序的正确性。

10.5.2　读写 EEPROM 例程

本小节将以 TI 的例程为例介绍 EEPROM 固件库的使用方法。

1）读写 EEPROM 程序。

```
 ******************************************************************************
//!文件名:write_eeprom. c
//!来源:  根据 TI 例程改编
//!功能描述:读写 EEPROM 中的数据并在 PuTTY 中显示出来
 ******************************************************************************
#include  < stdint. h >
```

```
#include  < stdbool. h >
#include " inc/hw_types. h"
#include " inc/hw_memmap. h"
#include " driverlib/sysctl. h"
#include " driverlib/pin_map. h"
#include " driverlib/debug. h"
#include " driverlib/gpio. h"
#include " driverlib/flash. h"
#include " driverlib/eeprom. h"
#include " driverlib/uart. h"
#include " utils/uartstdio. h"

// ****************************************************************************
//!初始化控制台函数
// ****************************************************************************
void
InitConsole( void)
{
    //
    //使能 GPIOA 的 UART0 引脚
    //
    SysCtlPeripheralEnable( SYSCTL_PERIPH_GPIOA) ;
    //
    //配置复用引脚 A0 和 A1 为 UART0 功能
    //
    GPIOPinConfigure( GPIO_PA0_U0RX) ;
    GPIOPinConfigure( GPIO_PA1_U0TX) ;
    //
    //使能 UART0 以便可以配置时钟
    //
    SysCtlPeripheralEnable( SYSCTL_PERIPH_UART0) ;
    //
    //使用内部 16 MHz 振荡器作为 UART 时钟源
    //
    UARTClockSourceSet( UART0_BASE,UART_CLOCK_PIOSC) ;
    //
    //选择这些引脚的复用(UART)功能
    //
    GPIOPinTypeUART( GPIO_PORTA_BASE,GPIO_PIN_0 | GPIO_PIN_1) ;
    //
    //初始化 UART 的控制台 I/O
    //
    UARTStdioConfig(0,115200,16000000) ;
}

int main( void)
{
    uint32_t pui32Data[2] ;
    uint32_t pui32Read[2] ;
```

```
        pui32Data[0] = 0x12345678;
        pui32Data[1] = 0x98abcde;
        //配置系统时钟频率为 40MHz
        SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|
                      SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

        //使能 EEPROM 外设
        SysCtlPeripheralEnable(SYSCTL_PERIPH_EEPROM0);
        //初始化 EEPROM
        EEPROMInit();

        //块擦除 EEPROM
        //注意:EEPROM 与 flash 不同,在写入前可以不擦除其中的内容,即这步视情况可省略
        EEPROMMassErase();

        //读出块擦除后的 EEPROM 中的数据
        EEPROMRead(pui32Read,0x0,sizeof(pui32Read));

        //在 PuTTY 中显示待写入闪存中的数据
        InitConsole();
        UARTprintf("**例程中待写入和从 EEPROM 中读回的数据**\n\n");
        UARTprintf("显示写入的数据:");
        UARTprintf(" %02x",pui32Data[0]);
        UARTprintf(" %02x",pui32Data[1]);
        UARTprintf("\n");

        //向擦除后的 EEPROM 中写入数据
        EEPROMProgram(pui32Data,0x0,sizeof(pui32Data));

        //读出向 EEPROM 中写入的数据
        EEPROMRead(pui32Read,0x0,sizeof(pui32Read));

        //显示读回已写入到 EEPROM 中的数据
        UARTprintf("显示读出的数据:");
        UARTprintf(" %02x",pui32Read[0]);
        UARTprintf(" %02x",pui32Read[1]);
        UARTprintf("\n");

        //循环等待
        while(1)
        {
        }
    }
```

2）创建 write_eeprom 工程。

3）编译工程生成 . out 格式可执行文件。

4）导入 . out 文件到 LaunchPad 板中，并在表达式（Expressions）窗口中添加 pui32Data 和 pui32Read 两个变量，并把数据设置成 hex 格式，如图 10-7 所示。

图 10-7　添加待写入的数据和读回数据变量

5）在 LaunchPad（EK - TM4C123GXL）板上对程序进行调试。

① 在 EEPROMMassErase( )设置一个断点，以观察块擦除语句是否能擦除 EEPROM 中的数据，其单步执行结果如图 10-8 所示。

图 10-8　单步执行块擦除语句后的结果

② 在 EEPROMRead( )处设置一个断点，以观察从 EEPROM 中读回的数据是否等于写入的数据，其单步执行结果如图 10-9 所示。

图 10-9　单步执行从 EEPROM 中读回的数据

在图 10-9 可以看到从 EEPROM 中读出的数据等于写入的数据，这就验证了该段程序的正确性。

6）在 LaunchPad（EK－TM4C123GXL）板上对程序进行测试。打开 PuTTY “串口助手”全速运行程序，其测试结果如图 10-10 所示。

```
COM7 - PuTTY
**例程中待写入和从EEPROM中读回的数据**

显示写入的数据: 12345678 98abcde
显示读出的数据: 12345678 98abcde
```

图 10-10　write_eeprom. c 程序的运行结果

从 PuTTY 显示的结果来看，write_eeprom. c 程序实现了写入和读出写入数据的功能。这时还可以在表达式（Expressions）窗口查看 pui32Data 和 pui32Read 的显示结果（如图 10-11 所示）。

Variables　Expressions　Registers

Expression	Type	Value	
pui32Data	unsigned int[2]	0x200000E0 (Binary)	0x200000E0
[0]	unsigned int	0x12345678 (Hex)	0x200000E0
[1]	unsigned int	0x098ABCDE (Hex)	
pui32Read	unsigned int[2]	0x200000E8 (Hex)	
[0]	unsigned int	0x12345678 (Hex)	
[1]	unsigned int	0x098ABCDE (Hex)	0x200000EC
Add new exp			

写入的数据

读出的数据

图 10-11　在表达式窗口的写入数据和读出数据

从图 10-11 可看到，写入 EEPROM 的数据和从 EEPROM 读回的数据完全相同，这就在硬件平台上验证了程序功能的正确性。

第 11 章

通用定时器（GPTM）

通用定时器（General – Purpose Timer，GPT）用于对驱动定时器输入引脚的外部事件进行计数/定时。TM4C123GH6PM 通用定时器模块（GPTM）具有 6 个 16/32 位的 GPTM 模块与 6 个 32/64 位宽的 GPTM 。每个 16/32 位 GPTM 可提供 2 个 16 位的定时器/计数器（即 Timer A 和 Timer B），用户可将其配置成独立运行的定时器或事件计数器，或使之连接成一个 32 位定时器或一个 32 位实时时钟（RTC）。每个 32/64 位宽的 GPTM 不但可以为 Timer A 和 Timer B 提供 32 位定时器，而且用户还可将它们连接成一个 64 位定时器。

另外，定时器也可以用来触发 ADC 转换与 μDMA 传输 。所有通用定时器的 ADC 触发信号在到达 ADC 模块之前可对其进行或运算，因此仅需一个定时器就可触发 ADC 事件。

通用定时器模块是 Tiva C 系列微控制器一个可用的定时资源。与之相对应的其他定时器资源是指系统定时器（SysTick） 和 PWM 模块的 PWM 定时器。

定时器 API 为定时器模块提供了一组用于定时器的配置与控制、修改定时器/计数器的值与管理定时器的中断处理的函数。

该驱动程序包含在 Driverlib/ timer. c 中，Driverlib/ timer. h 包含了定时器 API 函数的定义。

本章的主要内容：

➢ 通用定时器单元
➢ GPTM 固件库函数（见附录 I）
➢ 例程



11.1 通用定时器单元

11.1.1 主要特点

通用定时器模块提供的功能选择如下：

1）16/32 位操作模式：

① 16/32 位可编程的单次定时器。

② 16/32 位可编程的周期定时器。

③ 包含 8 位预分频的 16 位通用定时器。

④ 当使用 32. 768 kHz 的外部时钟输入时，可作为 32 位的实时时钟。

⑤ 包含 8 位预分频的 16 位输入沿计数或定时捕获模式。

⑥ 包含 8 位预分频的 16 位 PWM 模式和可软件编程的 PWM 信号反相输出。

2）32/64 位操作模式：

① 32/64 位可编程的单次定时器。

② 32/64 位可编程的周期定时器。

③ 包含 16 位预分频的 32 位通用定时器。

④ 当使用 32.768 kHz 的外部时钟源时，可作为 64 位的实时时钟。

⑤ 包含 16 位预分频的 32 位输入沿计数或定时捕获模式。

⑥ 包含 16 位预分频的 32 位 PWM 模式和可软件编程的 PWM 信号反相输出。

3）增、减计数。

4）12 个 16/32 位捕获比较 PWM 引脚（CCP）。

5）12 个 32/64 位捕获比较 PWM 引脚（CCP）。

6）菊花链式的定时器模块允许一个定时器启动多个定时事件。

7）定时器同步允许选择的定时器在同一时钟周期开始计数。

8）ADC 事件触发。

9）在调试过程中，当 CPU 出现暂停标识时，用户可以中止定时器事件（不含 RTC 模 式）。

10）可确定从发出中断请求到进入中断服务程序所花的时间。

11）使用 μDMA 来高效传输数据：

① 每个定时器的专用通道。

② 定时器中断响应突发请求。

11.1.2 GPTM 模块框图

通用定时器的模块框如图 11-1 所示。

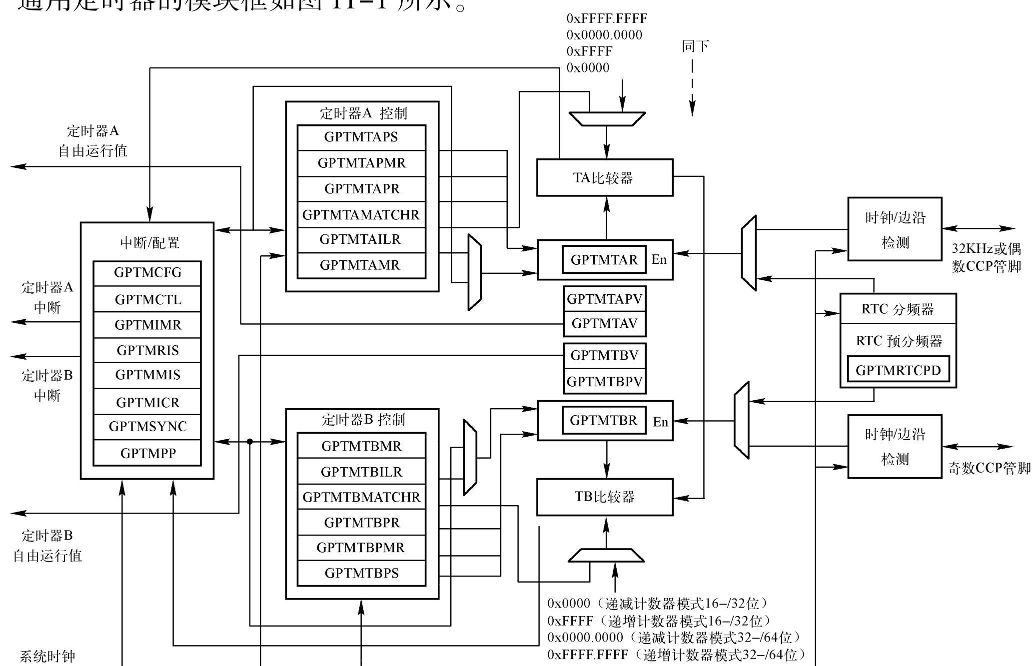


图 11-1　通用定时器的模块框图

11.1.3 信号描述

表 11-1 列出了 SSI 模块的外部信号，并描述了每个信号的功能。

表 11-1　通用定时器信号描述（64LQFP）

引脚名称	引脚编号	引脚复用/赋值	引脚类型	缓冲区类型	描　　述
T0CCP0	1 28	PB6(7) PF0(7)	I/O	TTL	16/32 位定时器 0 捕捉/比较/PWM 0
T0CCP1	4 29	PB7(7) PF1(7)	I/O	TTL	16/32 位定时器 0 捕捉/比较/PWM1
T1CCP0	30 58	PF2(7) PB4(7)	I/O	TTL	16/32 位定时器 1 捕捉/比较/PWM0
T1CCP1	31 57	PF3(7) PB5(7)	I/O	TTL	16/32 位定时器 1 捕捉/比较/PWM1
T2CCP0	5 45	PF4(7) PB0(7)	I/O	TTL	16/32 位定时器 2 捕捉/比较/PWM0
T2CCP1	46	PB1(7)	I/O	TTL	16/32 位定时器 2 捕捉/比较/PWM1
T3CCP0	47	PB2(7)	I/O	TTL	16/32 位定时器 3 捕捉/比较/PWM0
T3CCP1	48	PB3(7)	I/O	TTL	16/32 位定时器 3 捕捉/比较/PWM1
T4CCP0	52	PC0(7)	I/O	TTL	16/32 位定时器 4 捕捉/比较/PWM0
T4CCP1	51	PC1(7)	I/O	TTL	16/32 位定时器 4 捕捉/比较/PWM1
T5CCP0	50	PC2(7)	I/O	TTL	16/32 位定时器 5 捕捉/比较/PWM0
T5CCP1	49	PC3(7)	I/O	TTL	16/32 位定时器 5 捕捉/比较/PWM1
WT0CCP0	16	PC4(7)	I/O	TTL	32/64 位宽位定时器 0 捕捉/比较/PWM0
WT0CCP1	15	PC5(7)	I/O	TTL	32/64 位宽位定时器 0 捕捉/比较/PWM1
WT1CCP0	14	PC6(7)	I/O	TTL	32/64 位宽位定时器 1 捕捉/比较/PWM0
WT1CCP1	13	PC7(7)	I/O	TTL	32/64 位宽位定时器 1 捕捉/比较/PWM1
WT2CCP0	61	PD0(7)	I/O	TTL	32/64 位宽位定时器 2 捕捉/比较/PWM0
WT2CCP1	62	PD1(7)	I/O	TTL	32/64 位宽位定时器 2 捕捉/比较/PWM1
WT3CCP0	63	PD2(7)	I/O	TTL	32/64 位宽位定时器 3 捕捉/比较/PWM0
WT3CCP1	64	PD3(7)	I/O	TTL	32/64 位宽位定时器 3 捕捉/比较/PWM1
WT4CCP0	43	PD4(7)	I/O	TTL	32/64 位宽位定时器 4 捕捉/比较/PWM0
WT4CCP1	44	PD5(7)	I/O	TTL	32/64 位宽位定时器 4 捕捉/比较/PWM1
WT5CCP0	53	PD6(7)	I/O	TTL	32/64 位宽位定时器 5 捕捉/比较/PWM0
WT5CCP1	10	PD7(7)	I/O	TTL	32/64 位宽位定时器 5 捕捉/比较/PWM1

11.1.4 功能简介

每个 GPTM 的主要元件为 2 个自由运行的递增/递减计数器（称作 TimerA 和 TimerB）、2 个匹配寄存器、2 个预分频器匹配寄存器、2 个影子寄存器、2 个加载/初始化寄存器及其相关的控制功能。

定时器模块提供两个半宽的定时器/计数器，可配置成独立运行的定时器或事件计数器，或者配置成全宽定时器或实时时钟（RTC）。一些定时器提供了 16 位的半宽定时器和一个 32 位的全

宽定时器，而其他定时器提供了32位半宽定时器和一个64位的全宽定时器。API在一个定时器模块里提供两个半宽的定时器分别称为TimerA和TimerB，而全宽定时器称为TimerA。

当配置为全宽或半宽定时器时，定时器可以设置为单次触发定时器或一个连续的计时器。如果配置为单次模式，当定时器向下计数到0或者计数到装载值时，定时器就停止计数。如果配置为连续模式，定时器计数到0（向下计数）或装载值（向上计数）时，定时器将重新装载初值并继续计数。当配置为一个全宽的定时器时，该定时器也可以被配置成RTC。在这种模式下，定时器希望由一个32.768 kHz的外部时钟来驱动，该时钟被分频用于产生1 s的时钟节拍。

当在半宽模式下，定时器也可以配置为事件捕获或者作为脉冲宽度调制（PWM）发生器。当配置为事件捕获时，定时器作为计数器。它可配置成计数两个事件之间的时间或计数事件本身。被计数的事件可被配置成上升沿、下降沿或者双边沿。当定时器被配置为PWM发生器时，用于捕获事件的输入信号变为输出信号，定时器可用于驱动一个边沿对齐的脉冲到该信号上。

定时器模块还提供了能够控制其他功能的参数，如输出反转、输出触发、中止期间定时器的行为。

此外，定时器模块还提供了中断源和事件的控制。用生成的中断指示一个事件已捕获或者一定数量的事件已捕获。当定时器已向下计数到0或定时器匹配一个特定的值时也可产生中断。

TM4C1233H6PM还可以从多个定时器模块进行计数器同步。同步计数器对于PWM和边沿定时捕获模式较有帮助。在PWM模式下，PWM从多个定时器输出，可被具有相同的加载值锁步使其可同步计数。类似地，在边沿定时捕获模式下，使用相同的加载值和同步计数器，使两个输入边沿之间的绝对时间可很容易地测量。

表11-2列出各GPTM可用的模式。

表11-2　通用定时器功能

模式	定时器使用	计数方向	计数器大小		预分频器大小①		预分频器行为（计数器方向）
			16/32位GPTM	32/64位宽GPTM	16/32位GPTM	32/64位宽GPTM	
单次触发	独立	递增或递减	16位	32位	8位	16位	定时器扩展（递增） 预分频器（递减）
	级联	递增或递减	32位	64位	—	—	N/A
周期	独立	递增或递减	16位	32位	8位	16位	定时器扩展（递增） 预分频器（递减）
	级联	递增或递减	32位	64位	—	—	N/A
RTC	级联	递增	32位	64位	—	—	N/A
边沿计数	独立	递增或递减	16位	32位	8位	16位	定时器扩展（双向）
边沿定时	独立	递增或递减	16位	32位	8位	16位	定时器扩展（双向）
PWM	独立	递减	16位	32位	8位	16位	定时器扩展

① 仅在独立使用定时器时可使用预分频器。

注：

1. 在单次或周期模式下的递减计数时，预分频器用作真预分频器且包含计数的最低位。
2. 在单次或周期模式下的递增计数时，预分频器用作定时器扩展且包含计数的最高位。
3. 在输入沿计数、输入沿定时与PWM模式时，预分频器用作定时器扩展，而无需关心计数方向。

1. 定时器模式

本小节将简介几种常用定时器模式的运行机制。

(1) 单次触发/周期定时器模式

单次触发/周期定时器包括以下模式：

① 增/减计数模式。

② 等待触发模式。

③ 单次触发模式。当定时器满足条件后将停止计数并清除标志位。

④ 周期模式。计数器从 0 开始递增计数到装载值，重新自动装载初值，并从 0 开始重新计数；计数器从预装载值开始递减计数并重新装载预装载值。

⑤ μDMA 触发模式。

选择单次触发/周期模式可依据写入到模式寄存器（GPTMTnMR）中 TnMR 字段的值来确定，而定时器是递增计数还是递减计数由 GPTMTnMR 寄存器中的 TnCDIR 位来确定。

在软件将 GPTM 控制寄存器（GPTMCTL）中的 TnEN 位置位时，定时器将从 0x0 开始递增计数或从预装载的值开始递减计数。此外，若 GPTMTnMR 寄存器中的 TnWOT 位被置位，在 TnEN 位置位时，定时器将会等待一个触发来启动计数。在使能定时器时将要装载到定时器寄存器的值见表 11-3。

表 11-3　使能定时器时寄存器的值。

寄　存　器	递减计数模式	递增计数模式
GPTMTnR	GPTMTnILR	0x0
GPTMTnV	GPTMTnILR	0x0
GPTMTnPS	独立模式：GPTMTnPR；级联模式不可用	独立模式为 0x0；级联模式不可用
GPTMTnPV	独立模式：GPTMTnPR；级联模式不可用	独立模式为 0x0；级联模式不可用

当定时器做递减计数并到达超时事件（0x0）时，定时器将在下一个时钟周期从 GPTMTnILR 与 GPTMTnPR 寄存器中重新装载初值。当定时器做递增计数并达到超时事件（GPTMTnILR 和可选 GPTMTnPR 寄存器中的数值）时，定时器将会重新装载 0x0。若将定时器配置为单次触发模式，则定时器将停止计数，并将 GPTMCTL 寄存器中的 TnEN 位清零；如果将定时器配置为周期模式，则定时器将在下一个时钟周期重新开始计数。

在周期、快照模式（即 GPTMTnMR 寄存器中的 TnMR 字段为 0x2，TnSNAPS 位 = 1）发生超时事件时，定时器的值将被加载到 GPTMTnR 寄存器，而预分频器的值会被加载到 GPTMTnPS 寄存器中。自由运行计数器的值将显示在 GPTMTnV 寄存器中，自由运行预分频器的值会显示在 GPTMTnPV 寄存器中。通过此方式，软件可校验快照值和自由运行定时器的当前值，使之可确定从中断发生到进入中断处理程序之间所花费的时间。注意，当定时器配置为单次触发模式时，快照模式不可用。

除重载初值外，在定时器达到超时事件时将会被触发并产生超时中断；当 GPTMT-

nMR 寄存器中的 TnMIE 位置位时，在定时器的值与加载到匹配寄存器（GPTMTnMATCHR）和预分频匹配值寄存器（GPTMTnPMR）的值相等时，也将产生中断条件。通过将 GPTMCTL 中的 TnOTE 位置位可启动 ADC 触发，当 ADC 触发器使能时，仅单次触发或周期超时事件使 ADC 触发器有效。可通过配置并使能相应的 μDMA 通道来开启 μDMA 触发器。

若软件在计数器递减计数时更新了 GPTMTnILR 或 GPTMTnPR 寄存器，计数器将在下一个时钟周期装载新值；若 GPTMTnMR 寄存器中的 TnILD 位被清零，则计数器将从新值开始继续计数；如果 TnILD 位被置位，计数器将在下一个超时事件后加载新值；如果软件在计数器递增计数时更新了 GPTMTnILR 或 GPTMTnPR 寄存器，超时事件会在下一个时钟周期变更为新值；如果软件在计数器递增或递减计数时更新了 GPTM Timer n 值寄存器（GPTMTnV）的值，计数器将会在下个时钟周期载入这个新值并从该值开始递增或递减计数；若软件更新了 GPTMTnMATCHR 或 GPTMTnPMR 寄存器，新值会在下一个时钟周期反映出来；如果 TnMRSU 位被置位，新值将在下一个超时事件后才会生效。

（2）实时时钟模式

实时时钟模式仅支持递增计数模式，复位后计数器的值为 0x01，见表 11-4。

表 11-4　在 RTC 模式下的计数器值

寄　存　器	递减计数模式	递增计数模式
GPTMTnR	不可用	0x1
GPTMTnV	不可用	0x1
GPTMTnPS	不可用	不可用
GPTMTnPV	不可用	不可用

在 RTC 模式中，要求 CCP0 引脚的输入时钟为 32.768 kHz。然后将其分频为 1 Hz。在软件写 GPTMCTL 寄存器中的 TAEN 位时，计数器将从预装载值 0x1 开始递增计数。当计数器值与 GPTMTAMATCHR 寄存器中预装载值匹配时，GPTM 将使 GPTMRIS 寄存器中的 RTCRIS 位生效，并开始继续计数，直到发生硬件复位或被软件禁止（通过清零 TAEN 位）。当定时器的值达到最终计数值时，定时器将会返回到 0x0 继续递增计数。如果在 GPTMIMR 寄存器中使能 RTC 中断，则 GPTM 还将置位 GPTMMIS 寄存器中的 RTCMIS 位并产生控制器中断。通过写 GPTMICR 寄存器中的 RTCCINT 位可将状态标志清除。

为了确保 RTC 值的一致性，其软件流程如图 11-2 所示。

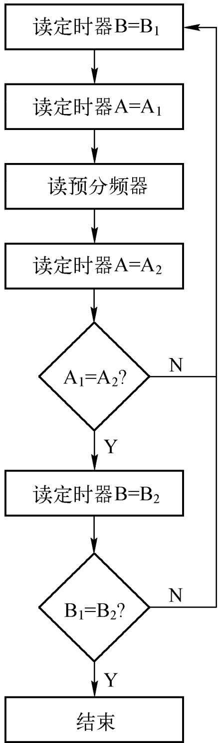


图 11-2　读 RTC 值的流程图

（3）输入边沿计数模式

在该模式中，最大输入频率为 1/4 系统频率，定时器配置为 24 位或 48 位增/减计数器，包括带有存储在通用定时器 n 中的预分频寄存器（GPTMTnPR）中的高计数值和 GPTMTnR 寄存器中低位的可选预分频器。在该模式下，定时器可捕捉到三种事件类

型：上升沿、下降沿或双沿。为了把定时器设置为边沿计数模式，GPTMTnMR 寄存器中的 TnCMR 位必须清零。定时器计数时所采用的边沿类型由 GPTMCTL 寄存器中的 TnEVENT 字段决定。在递减计数模式的初始化期间，需配置 GPTMTnMATCHR 和 GPTMTnPMR 寄存器，使 GPTMTnILR 和 GPTMTnPR 寄存器与 GPTMTnMATCHR 和 GPTMTnPMR 寄存器之间的差值等于待计算的边沿事件的个数。在递增计数模式中，定时器从 0x0 开始计数直到 GPTMTnMATCHR 和 GPTMTnPMR 寄存器中的值见表 11-5。

表 11-5　当定时器使能时输入边沿计数模式的计数器值

寄　存　器	递减计数模式	递增计数模式
GPTMTnR	GPTMTnILR	0x0
GPTMTnV	GPTMTnILR	0x0
GPTMTnPS	GPTMTnPR	0x0
GPTMTnPV	GPTMTnPR	0x0

在软件写 GPTM 控制寄存器（GPTMCTL）的 TnEN 位时，使能定时器以捕获事件。CCP 引脚每输入一个事件，计数器的值就递减 1，直到事件计数的值与 GPTMTnMATCHR 和 GPTMTnPMR 的值匹配。这时 GPTM 将让 GPTMRIS 寄存器确认 CnMRIS 位（若中断未被屏蔽，可同时确认 CnMMIS 位）。

此外，它还可以产生 ADC 和/或 μDMA 触发。在达到匹配值后，计数器将使用 GPTMTnILR 和 GPTMTnPR 寄存器中的值来执行重新装载操作，并由 GPTM 自动将 GPTMCTL 寄存器中的 TnEN 位清零，停止计数器计数。一旦事件计数值满足要求，随后的所有事件都将被忽略，直到通过软件重新将 TnEN 位使能。

例如，计数器配置为检测输入信号的双边沿，定时器的初值为 GPTMTnILR = 0x000A，匹配值为 GPTMTnMATCHR = 0x0006。此时，仅需计数四个边沿事件，如图 11-3 所示。

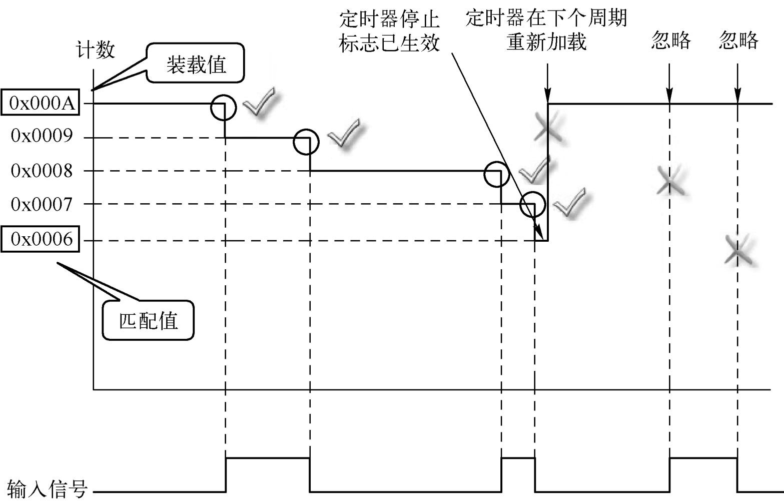


图 11-3　输入边沿计数模式示例

（4）输入边沿定时模式

在边沿定时模式中，最大输入频率为系统频率的 1/4，定时器配置为 24 位或 48 位递增/递减计数模式。对于该模式，在递减计数时需初始化装载值到 GPTMTnILR 寄存器中。定时器可捕获三种事件类型：上升沿、下降沿、或双沿。通过置位 GPTMTnMR 寄存器的 TnCMR 位可将定时器配置为边沿定时模式，而定时器捕获时采用的事件类型可由 GPTMCTL 寄存器的 TnEVENT 字段来决定。表 11-6 列出了输入事件计数模式的计数器值。

表 11-6　当定时器使能时输入事件计数模式的计数器值

寄　存　器	递减计数模式	递增计数模式
GPTMTnR	GPTMTnILR	0x0
GPTMTnV	GPTMTnILR	0x0
GPTMTnPS	GPTMTnPR	0x0
GPTMTnPV	GPTMTnPR	0x0

在软件写 GPTMCTL 寄存器中的 TnEN 位时，可使能定时器来捕获事件。在检测到所选输入事件时，从 GPTMTnR 和 GPTMTnPS 寄存器中捕获定时器计数器的当前值，且该值可通过微控制器来读取。然后 GPTM 将确认 CnERIS 位（若中断未被屏蔽，可同时确认 CnEMIS 位）。GPTMTnV 包含定时器的自由运行值，读取该值可以确定从中断发生到进入中断处理程序之间所花费的时间。

此外，它还可产生 ADC 或 μDMA 触发。在捕获到事件之后，定时器不会停止计数，而是继续计数，直到 TnEN 位被清零为止。当定时器达到超时值时，在递增模式中将重新装载 0x0，而在递减模式下需重新装载来自 GPTMTnILR 寄存器中的初值。

输入边沿定时模式的工作原理如图 11-4 所示。假设定时器的初值为默认值 0xFFFF，定时器配置为捕获上升沿事件。每当检测到上升沿事件时，当前计数值便加载到 GPTMTnR 寄存器中，该值将一直保持到在寄存器中检测到下一个上升沿（在此上升沿处，新的计数值被加载到 GPTMTnR 寄存器中）。

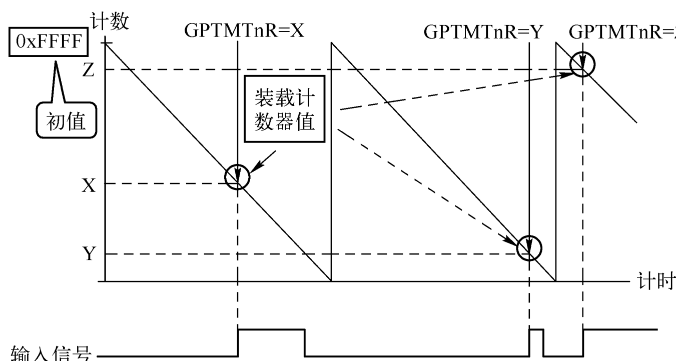


图 11-4　输入边沿定时模式的工作原理

（5）PWM 模式

GPTM 支持简单的 PWM 生成模式。在 PWM 模式中，将定时器配置为 24 位或 48 位递减计数器，初值由 GPTMTnILR 寄存器确定。在此模式中，PWM 频率和周期为同步事件，以

保障无毛刺现象。使 GPTMTnMR 寄存器中的 TnAMS 位 = 1、TnCMR 位 = 0、TnMR 字段 = 2 即可使能 PWM 模式。表 11-7 列出了 PWM 模式的计数器值。

表 11-7　当定时器使能时在 PWM 模式的计数器值

寄　存　器	递减计数模式	递增计数模式
GPTMTnR	GPTMTnILR	不可用
GPTMTnV	GPTMTnILR	不可用
GPTMTnPS	GPTMTnPR	不可用
GPTMTnPV	GPTMTnPR	不可用

在软件写 GPTMCTL 寄存器的 TnEN 位时，计数器开始递减计数，直到计数值到达 0x0 为止。如果 GPTMTnMR 寄存器中的 TnWOT 位置位，在 TnEN 位置位时，定时器将等待一个触发来启动计数。在周期模式的下一个计数循环中，计数器会将其初值重新装载到 GPTMTnILR 和 GPTMTnPR 寄存器中，并继续计数，直到用户通过软件将 GPTMCTL 寄存器中的 TnEN 位清零禁止计数。定时器可根据上升沿、下降沿或双沿这三种事件来产生中断。事件可通过 GPTMCTL 寄存器中的 TnEVENT 字段来配置，而中断可通过设置 GPTMTnMR 寄存器中的 TnPWMIE 位来使能。在发生该事件时，将使 GPTM 原始中断状态寄存器（GPTMRIS）中的 CnERIS 位置位，并保持该值直到在 GPTM 中断清除寄存器（GPTMICR）中执行写操作使之清零为止。若在 GPTM 中断屏蔽寄存器（GPTMIMR）中使能捕获事件中断，GPTM 还会将 GPTM 屏蔽的中断状态寄存器（GPTMMIS）中的 CnEMIS 位置位。注意，这里中断状态位不会被更新，除非 TnPWMIE 位被置位。

当计数器的值与 GPTMTnILR 和 GPTMTnPR 寄存器的值（计数器的初始状态）相等时，输出 PWM 信号有效，当计数器的值与 GPTMTnMATCHR 和 GPTMTnPMR 寄存器的值相等时，输出 PWM 信号无效。在 TnPWML 位 = 1 的条件下，可软件实现将输出 PWM 信号反相的功能。

例如，假设初值为 GPTMTnILR = 0xC350，匹配值为 GPTMTnMATCHR = 0x411A，如图 11-5 所示。

说明：从图 11-5 中可以看到当递减计数器的计数值为 0 时，将重新装载初值（GPTMTnILR = 0xC350），同时输出的 PWM 信号将翻转一次；当计数器的值等于设定的匹配值（GPTMTnMATCHR = 0x411A）时也将翻转一次。通过合理的设置初值和匹配值的大小可获得不同占空比的 PWM 信号。

2. 等待触发模式

等待触发模式允许使用菊花链式的定时器模块。若进行恰当的配置，一个单独的定时器可使用定时器触发来初始化多路时钟事件。将 GPTMTnMR 寄存器中的 TnWOT 位置位来使能等待触发模式。在 TnWOT 位置位时，Timer N + 1 只有等到菊花链中它的上一个（Timer N）达到超时事件时，才会开始计数。菊花链一般地配置形式为 GPTM1→GPTM0、GPTM2→GPTM1 等。如果 Timer A 在 32 位模式中，将触发下一个定时器模块的 Timer A。若 Timer A 在 16 位模式下，则将触发同一个定时器模块的 TimerB，而在下一个模块中，Timer B 将会触发 Timer A。注意：GPTM0 的 TAWOT 位永远不会置位。GPTMCFG 位如何影响菊花链如图 11-6 所示。

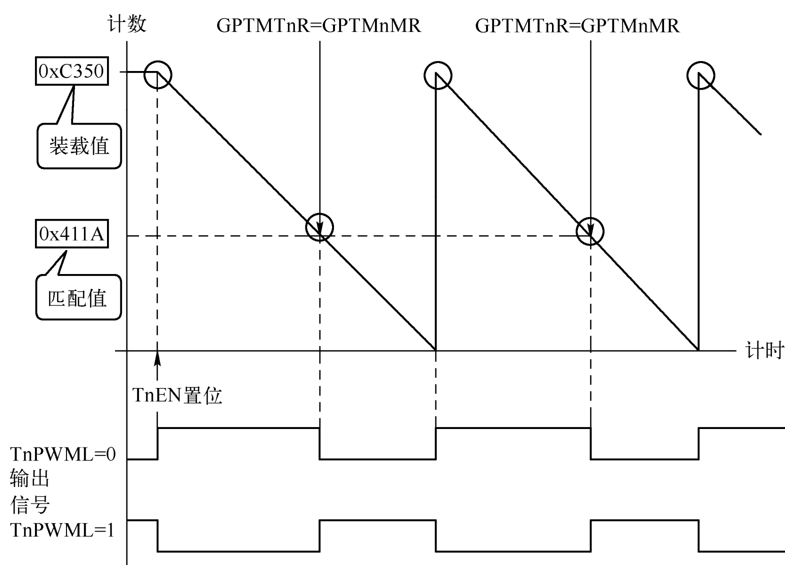


图 11-5 PWM 模式示例

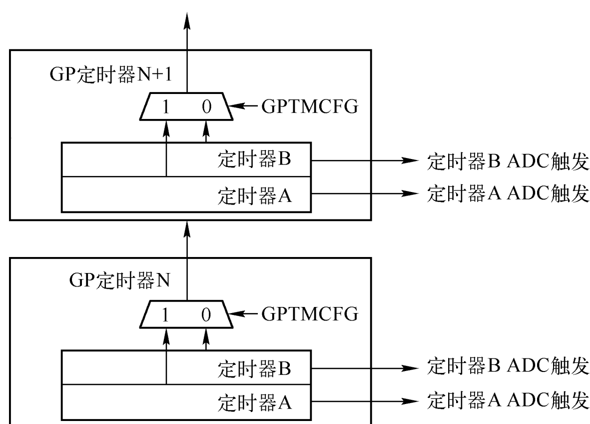


图 11-6　定时器菊花链

3. 同步通用定时器模块

GPTM 同步控制寄存器（GPTMSYNC）中的 GPTM0 模块可用于所选定时器的同步计数。设置 GPTMSYNC 寄存器中的位使相关的定时器用于处理超时事件的动作。而对于用于级联模式的计时器，只有定时器 A 中的位须在 GPTMSYNC 寄存器进行设置。在同步计数时，对定时器的超时操作，见表 11-8。

表 11-8　GPTM 模块的超时操作

模　　式	计数方向	超时行为
32 和 64 位单次	—	N/A
32 和 64 位周期	递减	计数值 = ILR
	递增	计数值 = 0

（续）

模　　式	计数方向	超时行为
32 和 64 位 RTC	递增	计数值 = 0
16 和 32 位单次	—	N/A
16 和 32 位周期	递减	计数值 = ILR
	递增	计数值 = 0
16 和 32 位边沿计数	递减	计数值 = ILR
	递增	计数值 = 0
16 和 32 位边沿定时	递减	计数值 = ILR
	递增	计数值 = 0
16 和 32 位 PWM	递减	计数值 = ILR

注：只有所有定时器使用相同的时钟源，才能使其正常工作。

4. DMA 操作

每个定时器都有一个专用的 μDMA 通道，并且还可为 μDMA 控制器提供一个突发型的请求信号，它在定时器原始中断条件出现时就会产生，μDMA 传输的仲裁个数应和每次定时器事件出现时需传输的数据单元个数相同，这些都无需要其他特殊的设置就可使能定时器用于 μDMA 操作。



11.2 GPTM 固件库函数

定时器的 API 可分成 3 个函数组：

① 处理定时器的配置。

② 处理定时器的内容。

③ 处理中断。

详细的 GPTM 固件库函数简介请见书后附录 I。



11.3 例程

下面将以 TI 提供周期定时器例程 timers. c 为例来介绍通用定时器固件库的使用、编程及测试方法。

1. 基于 EK - TM4C123GXL 板卡的周期定时器例程测试

1）定时器例程电路连线图，如图 11-7 所示。

2）timers. c 介绍。

```
//*****************************************************************************
//!文件名:timers. c
//!来源:TI 例程
//!功能描述:该例程介绍了如何使用定时器产生周期性中断。第一个定时器设置为每秒
//!产生一次中断,第二个定时器设置为每秒产生两次中断;每个中断处理器都会翻转自己
//!的指示器,并通过虚拟串行端口在“串口助手”PuTTY 上显示出来
```

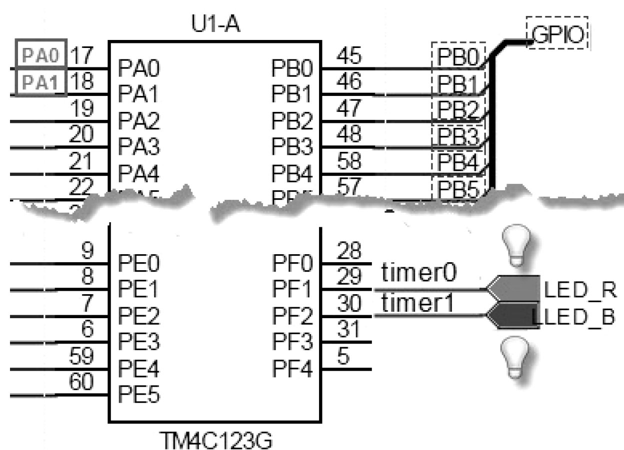


图 11-7　定时器例程电路连线图

```
// ****************************************************************************

#include  < stdint. h >
#include  < stdbool. h >
#include "inc/hw_ints. h"
#include "inc/hw_memmap. h"
#include "inc/hw_types. h"
#include "driverlib/debug. h"
#include "driverlib/fpu. h"
#include "driverlib/gpio. h"
#include "driverlib/interrupt. h"
#include "driverlib/pin_map. h"
#include "driverlib/rom. h"
#include "driverlib/sysctl. h"
#include "driverlib/timer. h"
#include "driverlib/uart. h"
#include "utils/uartstdio. h"

// ****************************************************************************
//设置一个全局变量用于当前中断状态的显示标志
// ****************************************************************************
uint32_t g_ui32Flags;

// ****************************************************************************
//发生固件库错误时调用此错误处理例程
// ****************************************************************************
#ifdef DEBUG
void
__error__(char  * pcFilename,uint32_t ui32Line)
{
}
#endif
```

```
// *****************************************************************************
//第一个定时器中断的中断处理程序
// *****************************************************************************
void
Timer0IntHandler(void)
{
    char cOne,cTwo;

    //
    //清除定时器中断
    //
    ROM_TimerIntClear(TIMER0_BASE,TIMER_TIMA_TIMEOUT);

    //
    //反转第一个计时器的标志
    //
    HWREGBITW(&g_ui32Flags,0) ^= 1;

    //
    //使用显示标志来反转定时器的指示 LED
    //
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1,g_ui32Flags << 1);

    //
    //更新 PuTTY 上的中断状态
    //
    ROM_IntMasterDisable();
    cOne = HWREGBITW(&g_ui32Flags,0) ? '1' : '0';
    cTwo = HWREGBITW(&g_ui32Flags,1) ? '1' : '0';
UARTprintf("\rT1: %c  T2: %c",cOne,cTwo);
    ROM_IntMasterEnable();
}

// *****************************************************************************
//第二个定时器中断的中断处理程序
// *****************************************************************************
void
Timer1IntHandler(void)
{
    char cOne,cTwo;

    //
    //清除定时器中断
    //
    ROM_TimerIntClear(TIMER1_BASE,TIMER_TIMA_TIMEOUT);

    //
    //反转第二个计时器的标志
    //
    HWREGBITW(&g_ui32Flags,1) ^= 1;

    //
```

```
    //使用显示标志来反转定时器的指示 LED
    //
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2,g_ui32Flags << 1);

    //
    //更新 PuTTY 上的中断状态
    //
    ROM_IntMasterDisable();
    cOne = HWREGBITW(&g_ui32Flags,0) ? '1' : '0';
    cTwo = HWREGBITW(&g_ui32Flags,1) ? '1' : '0';
    UARTprintf("\rT1: %c  T2: %c",cOne,cTwo);
    ROM_IntMasterEnable();
}

//*****************************************************************************
//配置 UART 与相应引脚
//*****************************************************************************
void
ConfigureUART(void)
{
    //
    //使能用于 UART 的 GPIO 外设
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    //
    //使能 UART0
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    //
    //将 GPIO 引脚配置为 UART 模式
    //
    ROM_GPIOPinConfigure(GPIO_PA0_U0RX);
    ROM_GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE,GPIO_PIN_0 | GPIO_PIN_1);

    //
    //使用内部 16 MHz 振荡器作为 UART 时钟源
    //
    UARTClockSourceSet(UART0_BASE,UART_CLOCK_PIOSC);

    //
    //初始化 UART 的控制台 I/O
    //
    UARTStdioConfig(0,115200,16000000);
}

//*****************************************************************************
//这个例程将介绍如何使用定时器来产生周期性中断
//*****************************************************************************
int
```

```
main(void)
{
    //
    //使能扩展(lazy)堆栈的中断处理程序
    //这可使浮点指令用于中断处理程序,但需要使用额外的堆栈空间
    //
    ROM_FPULazyStackingEnable();

    //
    //设置时钟直接从晶振(16 MHz)运行
    //
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                                       SYSCTL_XTAL_16MHZ);

    //
    //初始化 UART 和写状态
    //
    ConfigureUART();

    UARTprintf("\033[2JTimers example\n");
    UARTprintf("T1: 0   T2: 0");

    //
    //  使能 LaunchPad 板上 LED 的 GPIO 端口
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,GPIO_PIN_2 | GPIO_PIN_1);

    //
    //使能 TIMER0 与 TIMER1 使用的外设
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);

    //
    //使能处理器中断
    //
    ROM_IntMasterEnable();

    //
    //配置 2 个 32 位周期定时器
```

```
    //
    ROM_TimerConfigure(TIMER0_BASE,TIMER_CFG_PERIODIC);
    ROM_TimerConfigure(TIMER1_BASE,TIMER_CFG_PERIODIC);

    //设置定时器 0 的装载值为 16M 次,即运行 16M 次所花的时间为 1 s
    ROM_TimerLoadSet(TIMER0_BASE,TIMER_A,ROM_SysCtlClockGet());

    //设置定时器 1 的装载值为 8M 次,即 0.5 s
    ROM_TimerLoadSet(TIMER1_BASE,TIMER_A,ROM_SysCtlClockGet() / 2);

    //
    //设置定时器 0 与定时器 1 为超时中断
    //
    ROM_IntEnable(INT_TIMER0A);
    ROM_IntEnable(INT_TIMER1A);
    ROM_TimerIntEnable(TIMER0_BASE,TIMER_TIMA_TIMEOUT);
    ROM_TimerIntEnable(TIMER1_BASE,TIMER_TIMA_TIMEOUT);

    //
    //使能定时器 0 与定时器 1
    //
    ROM_TimerEnable(TIMER0_BASE,TIMER_A);
    ROM_TimerEnable(TIMER1_BASE,TIMER_A);

    //
    //循环等待
    //
    while(1)
    {
    }
}
```

3）创建 timer 工程，添加 timer. c 文件，如图 11-8 所示。

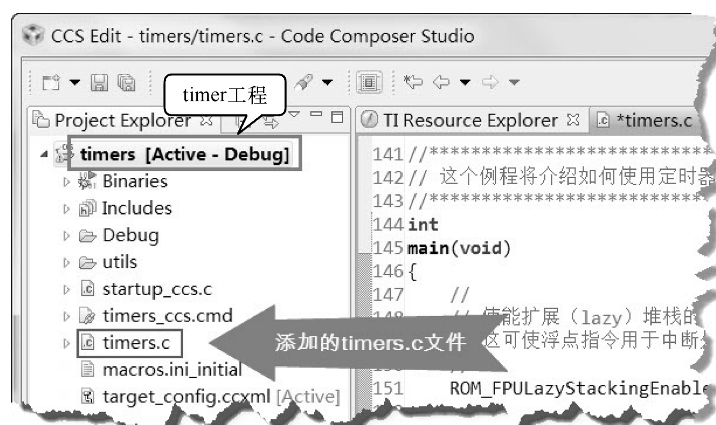


图 11-8　创建的 timer 工程

4）编译 timer 工程，并将其下载到 EK－TM4C123GXL 开发板中，如图 11-9 所示。

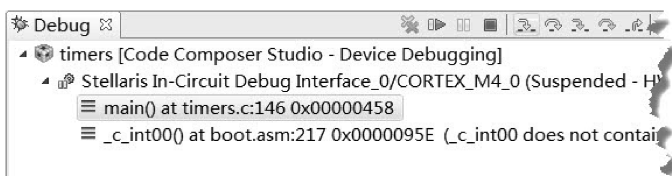


图 11-9　将编译获得的 . axf 可执行文件下载到 EK－TM4C123GXL 板中

5）测试。

① 打开 PuTTY 可看到如图 11－10 所示的信息。

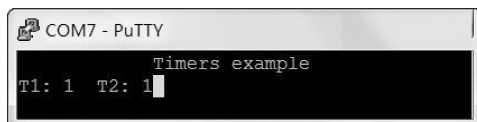


图 11-10　在 PuTTY 中显示的信息

说明：从图 11－10 中可以看到，T2 的数字变化速度比 T1 的数字变化速度快一倍（读者可以简单地用一个计时工具检验 T1 的定时器超时中断响应是否为 1 s?），满足 timer0 每秒产生一次中断、timer1 每秒产生两次中断的设计要求。

② 在 EK－TM4C123GXL 板中所观察到的测试结果：首先点亮的是蓝色 LED 灯（timer1 连接到了 LED_ B，见图 11-7 和表 11-1）；然后点亮红色 LED（timer0 连接到了 LED_ R，见图 11-7 和表 11-1）；最后同时点亮红色 LED 和蓝色 LED 灯。这是因为，Timer1 每秒产生两次定时器超时中断，而 Timer0 每秒仅产生一次定时器超时中断，所以会出现蓝色 LED 被首先点亮（熄灭）→红色 LED 点亮→红色 LED + 蓝色 LED 同时点亮的现象，这就进一步验证了上述程序的正确性。

2. 基于 Proteus 的周期定时器例程 timers 测试

1）周期定时器 timers 程序。

```
//*********************************************************************
//文件名:timers. c - Timers example
//来源:根据 TI 例程改编
//功能:周期定时器
//作用:供无实际板卡的读者测试定时器程序之用
//*********************************************************************
#include "inc/hw_ints. h"
#include "inc/hw_memmap. h"
#include "inc/hw_types. h"
#include "driverlib/debug. h"
#include "driverlib/gpio. h"
#include "driverlib/interrupt. h"
#include "driverlib/sysctl. h"
#include "driverlib/timer. h"
#include "drivers/pdc. h"
#include "utils/uartstdio. h"
//*********************************************************************
//
//如果驱动程序库遇到错误将调用错误处理程序
//
```

```
//*****************************************************************************
#ifdef DEBUG
void
__error__(char *pcFilename,unsigned long ulLine)
{
}
#endif
//*****************************************************************************
//
//定时器 0 中断处理程序
//
//*****************************************************************************
void
Timer0IntHandler(void)
{
//
    //清除中断
    //
    TimerIntClear(TIMER0_BASE,TIMER_TIMA_TIMEOUT);
    //
    //反转 GPIO B0
    //
    GPIOPinWrite(GPIO_PORTB_BASE,GPIO_PIN_0,
                 GPIOPinRead(GPIO_PORTB_BASE,GPIO_PIN_0) ^ GPIO_PIN_0);
}
//*****************************************************************************
//
//定时器 1 中断处理程序
//
//*****************************************************************************
void
Timer1IntHandler(void)
{
    //
    //清除定时器中断
    //
    TimerIntClear(TIMER1_BASE,TIMER_TIMA_TIMEOUT);
    //
    //反转 GPIO B1
    //
    GPIOPinWrite(GPIO_PORTB_BASE,GPIO_PIN_1,
                 GPIOPinRead(GPIO_PORTB_BASE,GPIO_PIN_1) ^ GPIO_PIN_1);
}
//*****************************************************************************
//
//配置 UART0 用于信息显示
//
//*****************************************************************************
void
```

```
InitConsole(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);
    GPIOPinTypeUART(GPIO_PORTA_BASE,GPIO_PIN_0 | GPIO_PIN_1);
    UARTStdioInit(0);
}
// ***********************************************************************
//
//主函数
//
// ***********************************************************************
int
main(void)
{
    //
    //设置系统时钟
    //
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_6MHZ);
    //
    //初始化 UART 及打印输出信息
    //
    InitConsole();
    UARTprintf("\Timers example\n");
    UARTprintf("---------------\n");
    UARTprintf("T1 = SysClock = LED_Y\n");
    UARTprintf("\T2 = (1/2)SysClock = LED_R\n");
    //
    //使能所用的外设
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    //
    //使能处理器中断
    //
    IntMasterEnable();
    //
    //将 GPIOB 端口的 B0 和 B1 设置为输出引脚
    //
    GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE,GPIO_PIN_0 | GPIO_PIN_1);
    GPIOPinWrite(GPIO_PORTB_BASE,GPIO_PIN_0 | GPIO_PIN_1,0);
    //
    //配置两个 32 位周期定时器
    //
```

```
    TimerConfigure(TIMER0_BASE,TIMER_CFG_32_BIT_PER);
    TimerConfigure(TIMER1_BASE,TIMER_CFG_32_BIT_PER);
    TimerLoadSet(TIMER0_BASE,TIMER_A,SysCtlClockGet());
    TimerLoadSet(TIMER1_BASE,TIMER_A,SysCtlClockGet() / 2);
    //
    //设置定时器超时中断
    //
    IntEnable(INT_TIMER0A);
    IntEnable(INT_TIMER1A);
    TimerIntEnable(TIMER0_BASE,TIMER_TIMA_TIMEOUT);
    TimerIntEnable(TIMER1_BASE,TIMER_TIMA_TIMEOUT);
    //
    //使能定时器
    //
    TimerEnable(TIMER0_BASE,TIMER_A);
    TimerEnable(TIMER1_BASE,TIMER_A);
    //
    //响应定时器中断
    //
    while(1)
    {
    }
}
```

2）搭建周期定时器程序的测试电路，如图 11-11 所示。

说明： 如果给出的测试电路不能运行，请先行加载文件夹 proteus_timers 中的 timers. elf 文件。

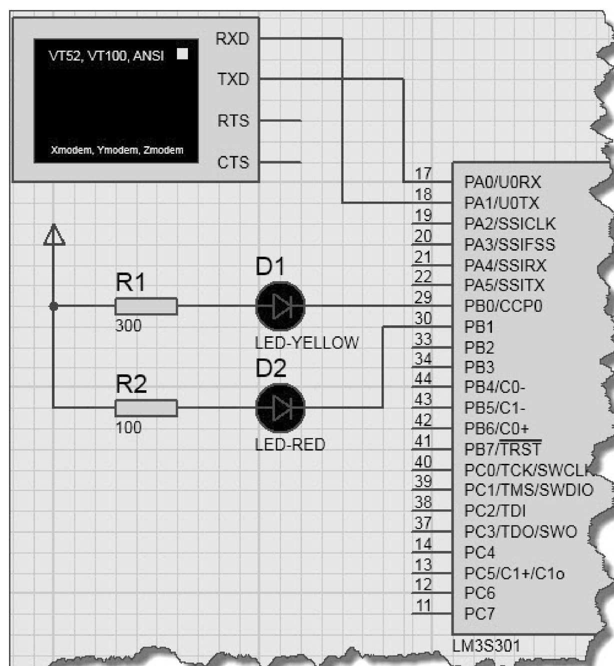


图 11-11　周期定时器程序的测试电路

3）测试结果如图 11-12 所示。

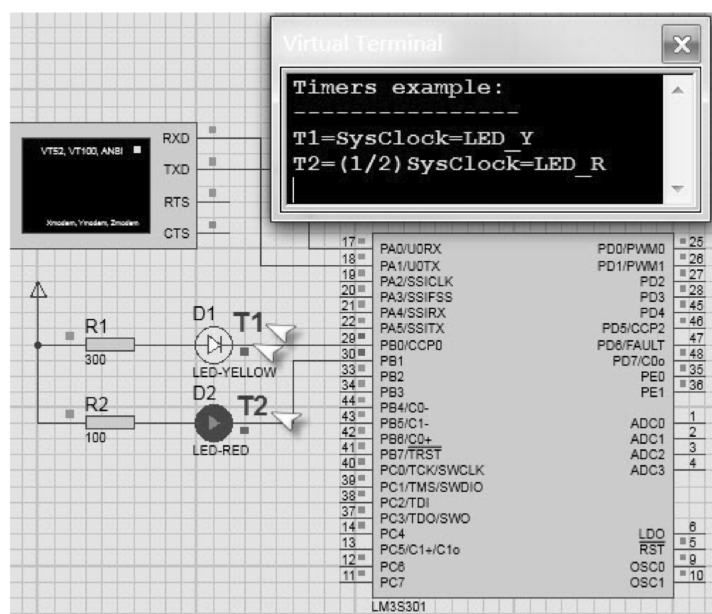


图 11-12　周期定时器程序测试结果

在图 11-12 测试电路中所观察到的现象和在 EK-TM4C123GXL 板卡中的现象一致。

说明： 如果给出的测试电路不能运行，请先行加载文件夹 proteus_timers 中的 timers. elf 文件。

第 12 章

脉冲宽度调制（PWM）

脉冲宽度调制（Pulse width modulation，PWM）是一项常用的将模拟信号电平进行数字编码的技术。在 PWM 中使用高分辨率计数器产生方波，并通过调控方波的占空比实现对模拟信号的编码，其典型应用为开关电源和运动控制。

TM4C123GH6PM 微控制器包含 2 个 PWM 模块，而每个模块又包含 4 个 PWM 发生器和 1 个控制模块，共 16 路 PWM 输出。控制模块决定了 PWM 信号的极性及哪路信号传递到引脚。

每个 PWM 发生器模块可产生 2 路 PWM 信号，这两路信号共享同一个定时器和频率，也可通过编程产生独立的信号，如插入死区延时的互补信号。PWM 发生器的输出信号为 pwmA′和 pwmB′。在发送到 PWM0 、PWM1 或者 PWM2 以及 PWM3 等引脚之前，这两个输出信号由输出控制模块管理。

TM4C123GH6PM PWM 模块提供大的灵活性，它不但可产生简单的 PWM 信号，如简易充电泵需要的信号；而且可产生带死区延迟的成对 PWM 信号，如半 – H 桥驱动。3 个发生器模块也可产生 3 相反相器桥所需的 6 通道栅极控制信号。

此驱动程序的 API 包含在 driverlib/pwm. c 中，而 driverlib/pwm. h 包含了应用程序使用的 API 的定义。

本章的主要内容：

- PWM 模块
- PWM 固件库函数（请见书后附录 J）
- 例程



12. 1　PWM 单元

12. 1. 1　PWM 的主要特点

1）每个 PWM 发生器模块具有以下特点：

① 1 个故障条件处理输入，能提供快速低延迟关闭，可防止电动机在被控前损坏，共 2 个输入。

② 1 个 16 位计数器。

- 运行在递减或先递增后递减模式。
- 输出频率由一个 16 位的装载值控制。
- 装载值可同步更新。
- 在零和装载值时产生输出信号。

③ 两个 PWM 比较器：

- 比较值可同步更新。
- 产生匹配输出信号。

④ PWM 信号发生器：

- 由计数结果所采取的行动和 PWM 的比较器输出信号构建 PWM 的输出信号。
- 产生两个独立的 PWM 信号。

⑤ 死区发生器：

- 产生 2 路死区延迟可编程的 PWM 信号以适合驱动半 H 桥。
- 可旁路，保留未修改的输入 PWM 信号。

⑥ 可以启动一个 ADC 采样序列。

2）PWM 控制块的选项如下：

① 每个 PWM 信号的 PWM 输出使能。

② 每个 PWM 信号输出的反相选择（极性控制）。

③ 每个 PWM 信号的故障处理选择。

④ PWM 发生器模块的定时器同步。

⑤ PWM 发生器模块间的定时器/比较器同步更新。

⑥ 扩展 PWM 发生器模块间的定时器/比较器同步更新。

⑦ PWM 发生器模块中断状态汇总。

⑧ 可编程极性和过滤，扩展 PWM 故障处理多个故障信号。

⑨ PWM 发生器可独立操作或者与其他发生器同步操作。

12.1.2 PWM 的模块框图

TM4C123GH6PM 控制器的模块框图包含两个 PWM 模块和 PWM 发生器，分别如图 12-1 和图 12-2 所示。

12.1.3 信号描述

PWM 模块的外部信号及功能描述见表 12-1。

12.1.4 功能简介

1. 时钟配置

PWM 具有两个可选择的时钟源：

① 系统时钟。

② 预分频系统时钟。

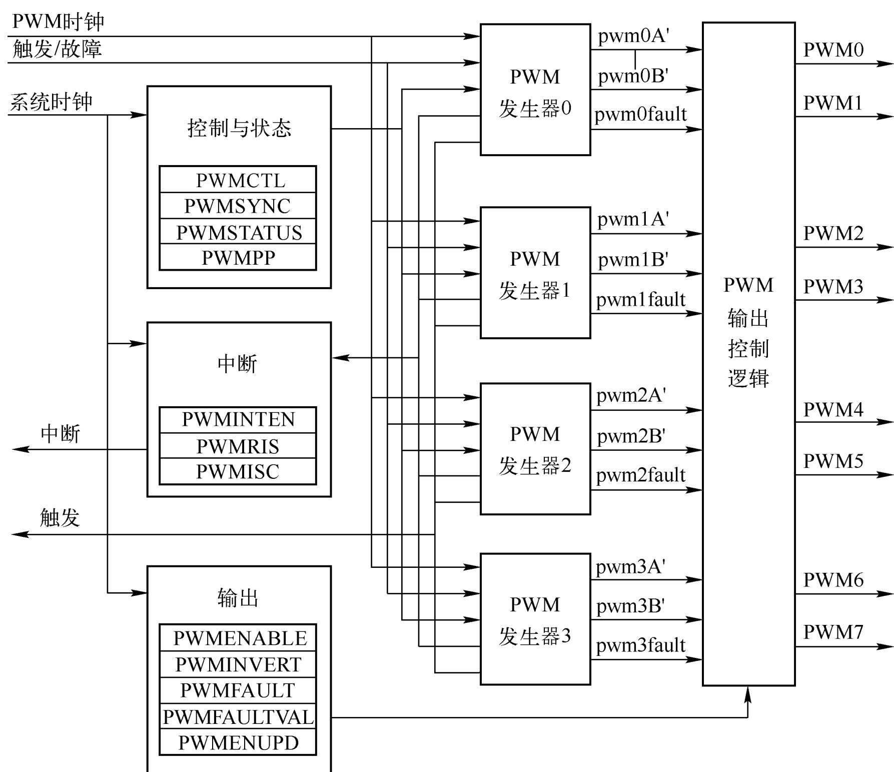


图 12-1　PWM 模块框图

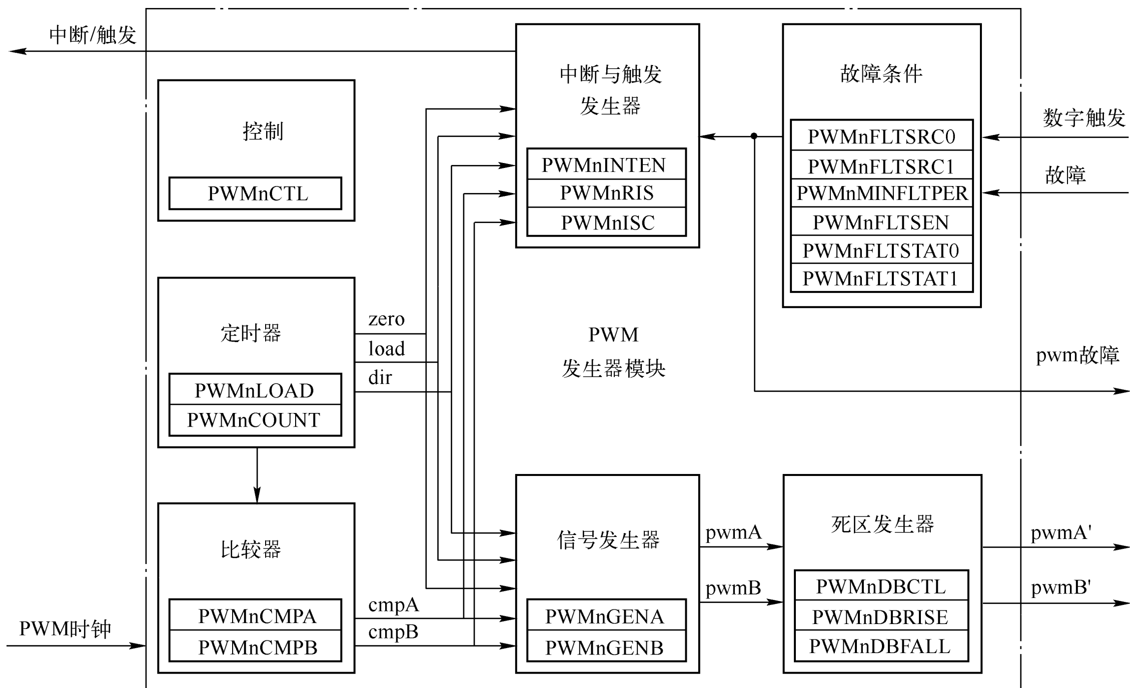


图 12-2　PWM 发生器模块框图

表 12-1　PWM 信号（64LQFP）

引脚名	引脚编号	引脚复用/赋值	引脚类型	缓冲器类型	描　　述
M0FAULT0	30 53 63	PF2（4） PD6（4） PD2（4）	I	TTL	运动控制模块 0 PWM 故障 0
M0PWM0	1	PB6（4）	O	TTL	运动控制模块 0 PWM 0（该信号由模块 0 PWM 发生器 0 控制）
M0PWM1	4	PB7（4）	O	TTL	运动控制模块 0 PWM1（该信号由模块 0 PWM 发生器 0 控制）
M0PWM2	58	PB4（4）	O	TTL	运动控制模块 0 PWM 2（该信号由模块 0 PWM 发生器 1 控制）
M0PWM3	57	PB5（4）	O	TTL	运动控制模块 0 PWM 3（该信号由模块 0 PWM 发生器 1 控制）
M0PWM4	59	PE4（4）	O	TTL	运动控制模块 0 PWM 4（该信号由模块 0 PWM 发生器 2 控制）
M0PWM5	60	PE5（4）	O	TTL	运动控制模块 0 PWM 5（该信号由模块 0 PWM 发生器 2 控制）
M0PWM6	16 61	PC4（4） PD0（4）	O	TTL	运动控制模块 0 PWM 6（该信号由模块 0 PWM 发生器 3 控制）
M0PWM7	15 62	PC5（4） PD1（4）	O	TTL	运动控制模块 0 PWM 7（该信号由模块 0 PWM 发生器 3 控制）
M1FAULT0	5	PF4（5）	I	TTL	运动控制模块 1 故障 0
M1PWM0	61	PD0（5）	O	TTL	运动控制模块 1 PWM 0（该信号由模块 1 PWM 发生器 0 控制）
M1PWM1	62	PD1（5）	O	TTL	运动控制模块 1 PWM 1（该信号由模块 1 PWM 发生器 0 控制）
M1PWM2	23 59	PA6（5） PE4（5）	O	TTL	运动控制模块 1 PWM 2（该信号由模块 1 PWM 发生器 1 控制）
M1PWM3	24 60	PA7（5） PE5（5）	O	TTL	运动控制模块 1 PWM 3（该信号由模块 1 PWM 发生器 1 控制）
M1PWM4	28	PF0（5）	O	TTL	运动控制模块 1 PWM 4（该信号由模块 1 PWM 发生器 2 控制）
M1PWM5	29	PF1（5）	O	TTL	运动控制模块 1 PWM 5（该信号由模块 1 PWM 发生器 2 控制）
M1PWM6	30	PF2（5）	O	TTL	运动控制模块 1 PWM 6（该信号由模块 1 PWM 发生器 3 控制）
M1PWM7	31	PF3（5）	O	TTL	运动控制模块 1 PWM 7（该信号由模块 1 PWM 发生器 3 控制）

2. PWM 定时器

每个 PWM 发生器中的定时器要么运行在递减模式，要么运行在先递增后递减模式。在递减模式中，定时器从装载值开始递减计数到零，然后返回到装载值，再递减计数。在先递增后递减模式中，定时器从零开始计数至装载值，然后递减到 0，再递增到装载值，循环往复。递减计数模式用于产生左或右对齐的 PWM 信号，而先递增后递减模式用于产生中心对齐的 PWM 信号。

定时器输出 PWM 生成过程中所使用的三个信号如下：

① "dir"（方向信号）。在递减计数模式中始终为低，而在先递增后递减模式中，其电平在低、高之间切换。

② "zero"（零脉冲信号）。当计数器的计数值为零时，零脉冲信号将产生一个单时钟周期宽的高电平脉冲。

③ "load"（装载脉冲）。当计数器的计数值等于装载值时，装载脉冲将产生一个单时钟周期宽高电平脉冲。（如图 12-3、图 12-4 所示）。

注意：*在递减计数模式中，零脉冲后面紧跟着装载脉冲。*

3. PWM 比较器

每个 PWM 发生器都有两个比较器，用于监控计数器的值；当比较器的值与计数器的值匹配时，比较器输出宽度为单时钟周期的高电平脉冲，即比较器输出一个宽度等于时钟周期的高电平脉冲，如图 12-3 和图 12-4 中“cmpA”与“cmpB”信号。在先递增后递减计数模式中，比较器在递增和递减计数时都需进行比较，因此必须采用计数器的方向信号来限定。这些限定脉冲将在产生 PWM 信号期间使用。如果比较器的值大于计数器的装载值，那么这个比较器将永远不会输出高电平脉冲，图 12-3 显示了在递减计数模式时的行为和这些脉冲的关系。图 12-4 给出了在先递增后递减计数模式下计数器的行为和这些脉冲之间的关系。

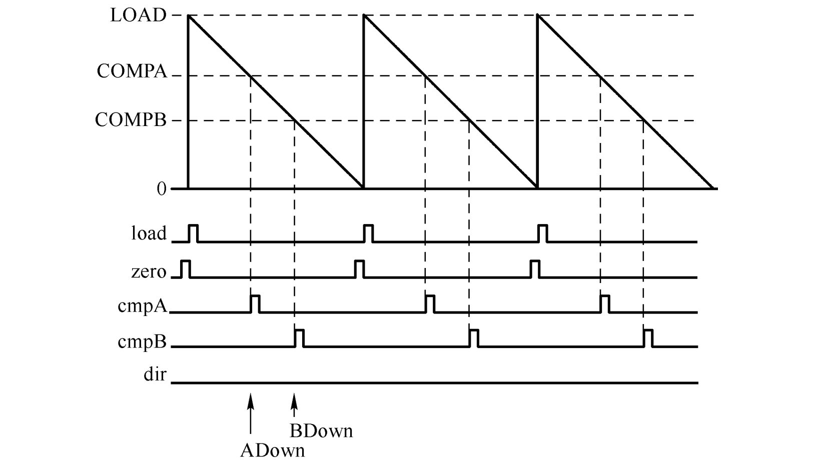


图 12-3　递减计数模式

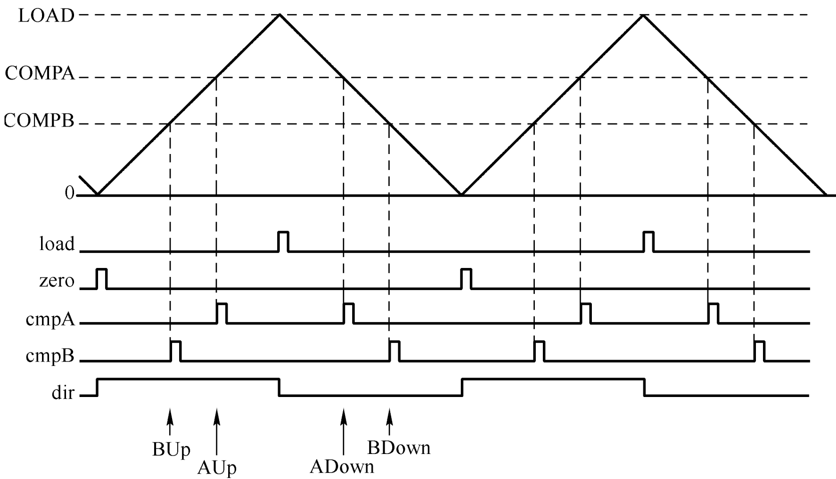


图 12-4　先递增后递减计数模式

4. PWM 信号发生器

PWM 发生器捕获 load、zero、cmpA 和 cmpB 脉冲（由方向信号限定），产生两个 PWM 信号（pwmA 和 pwmB）。在递减计数模式中，有 4 个事件可影响 PWM 信号：零、装载、匹配 A 递减、匹配 B 递减。在先递增后递减计数模式中，有 6 个事件可影响 PWM 信号：零、装载、匹配 A 递减、匹配 A 递增、匹配 B 递减、匹配 B 递增。当匹配 A 或匹配 B 事件与零或装载事件一致时，它们可被忽略。如果匹配 A 与匹配 B 事件相吻合，则第一个信号 PWMA 仅根据匹配 A 事件产生，第二个信号 pwmB 仅根据匹配 B 事件产生。

每个影响 PWM 输出信号的事件都是可编程的：可被保留（忽略该事件）、可被翻转、可被驱动为低/高电平。这些操作可用于产生一对不同位置和占空比的 PWM 信号，且该对信号既可重叠也可不重叠。图 12-5 给出了在先递增后递减计数模式下产生的一对中心对齐、含有不同占空比的重叠 PWM 信号。

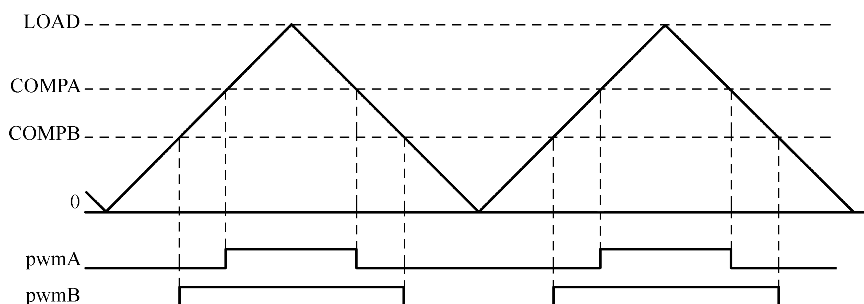


图 12-5　先递增后递减模式 PWM 信号发生图例

从图 12-5 中可以看到，第一个 PWM 发生器被设置为当发生匹配 A 递增事件时将其驱动为高电平，发生匹配 A 递减事件时使其为低电平，并忽略其他 4 个事件。第二个发生器被设置为当发生匹配 B 递增事件时将其驱动为高电平，发生匹配 B 递减事件时使其为低电平，并忽略其他 4 个事件。通过改变比较器 A 的值可改变 pwmA 信号的占空比，同样改变比较器 B 的值也可改变 pwmB 信号的占空比。

5. 死区发生器

PWM 发生器产生 pwmA 和 pwmB 两个 PWM 信号并传递给死区发生器。若禁止死区发生器，则 PWM 信号只简单地通过该模块得到 pwmA′和 pwmB′，而不会发生改变。如果使能死区发生器，将会丢弃 pwmB 信号，仅在 pwmA 信号基础上产生两个 PWM 信号。第一个输出 PWM 信号（pwmA′）为 pwmA 上升沿延迟可编程的信号。第二个输出 PWM 信号（pwmB′）与 pwmA 信号反相，为带延迟时间可编程的下降沿延迟信号如图 12-6 所示。

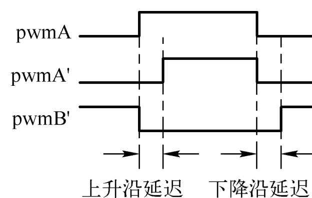


图 12-6　PWM 死区发生器

6. 中断/ADC－触发选择器

每个 PWM 发生器捕获相同的 4 个（或 6 个）计数器事件来产生中断或 ADC 触发事件。可选择这些事件中的任一个或一组作为中断源，只要所选定的事件其中一个发生时都将产生中断。此外，也可选择相同事件、不同事件、一组相同事件、一组不同事件作为 ADC 的触发源，当这些选定的任何事件发生时同样会产生 ADC 触发脉冲。选择的事件允许在 pwmA 或 pwmB 信号内的特定位置产生中断或触发 ADC 转换。

7. 同步方法

常用的 PWM 操作如下：

① 异步：PWM 发生器和它的两个输出信号可单独使用，且独立于其他 PWM 发生器。

② 同步：PWM 发生器和它的两个输出信号共享统一的时基与其他 PWM 发生器连接。如果多个 PWM 发生器都配置成了相同的计数器装载值，则同步可保证其也具有相同的计数值。有了这个特性，就可产生两个以上的 PWMn 信号，使这些信号的边沿之间包含某些已知的关系，并且模块中的其他状态也提供了一种维持共享时基和相互同步的机制。

若向 PWM 时间基准同步寄存器（PWMSYNC）中写入数据，可将 PWM 发生器中的计数器复位，以及将发生器相关的 SYNCn 位置位。并可在一次访问中置位所有需要的 SYNCn 位以同步多个 PWM 发生器。例如，若将 PWMSYNC 寄存器中的 SYNC0 和 SYNC1 位置 1 就可使 PWM 发生器 0 和 PWM 发生器 1 中的计数器同时复位。

8. 故障状态

故障状态时必须向控制器发出停止正常 PWM 功能的信号，并置位 MnPWMn 信号到安全状态。两种基本情况将会引发故障状态：

① 微控制器被中止且在运动控制中不能及时执行必要的计算。

② 检测到外部错误或事件。

每个 PWM 发生器可用下面的输入来产生故障状态：

① MnFAULTn 引脚有效。

② 由调试器引发的控制器中止。

③ ADC 数字比较器的触发。

故障状态的计算是以 PWM 发生器为基础的。每个 PWM 发生器需配置必要的条件来指示已存在的故障状态，允许开发程序可独立和从属的控制。

2 个故障输入引脚（MnFAULTn）可供选择。这些输入可用于和产生高/低电平有效的电路一起来指示一个错误状态。MnFAULTn 引脚可使用 PWMnFLTSEN 寄存器单独编程来得到适当的逻辑意义。

PWM 发生器的控制模式，包括故障状态处理，将在 PWMnCTL 寄存器中给出。该寄存器决定是输入信号还是 MnFAULTn 输入信号的组合，和/或使用数字比较器触发来产生一个故障状态。PWMnCTL 寄存器也可以选择故障状态是维持与外界状态一样长，还是被锁存直到由软件清除故障状态为止。最后，这个寄存器也使能一个可用于扩展故障状态周期的计数器，以确保外部事件持续时间最短。该最小故障周期计数在 PWMnMINFLTPER 寄存器中指定。

PWMnFLTSTAT0 和 PWMnFLTSTAT1 寄存器提供了与具体故障原因相关的状态。PWMINTEN 寄存器可用于将 PWM 发生器故障状态提升为控制器中断。

9. 输出控制块

在 pwmA′与 pwmB′作为 MnPWMn 信号到达引脚之前，输出控制块将负责对其进行最后的调整。通过 PWM 输出使能寄存器（ PWMENABLE)，可对实际到达引脚上的一组 PWM 信号进行修改。例如，此功能可用于执行单个寄存器的写操作来实现跟无刷直流电动机的通信（无需修改各个 PWM 发生器，只需对反馈控制回路的进行修改)。此外，通过 PWM 更新使能寄存器（PWMENUPD)，可对 PWMENABLE 寄存器中的位更新进行配置来立即、局部，或全局同步到下一个同步更新。

故障状态期间，PWM 输出信号 MnPWMn，通常必须被驱动为安全值，使外部设备可以安全地控制。PWMFAULT 寄存器指定在故障状态下是否继续驱动已产生的信号或在 PWM-FAULTVAL 寄存器中一个指定的编码。

每路 PWM 信号都有可配置的输出反向控制。即通过 PWM 输出反相寄存器（PWMINVE-RT）可使任何 MnPWMn 信号反相，使其从默认的高电平有效变成低电平有效。即使某值在 PWMFAULT 寄存器中使能并在 PWMFAULTVAL 寄存器中指定，仍可反相。



12.2 PWM 固件库函数

PMW 固件库函数执行 PWM 模块的高级操作。PWM 的固件函数为用户提供了一种方法，以便进行 PWM 模块的常见配置：如设置周期、产生左对齐和中心对齐的脉冲、修改脉宽、控制中断、触发和输出特性。PWM 模块可被配置成多种不同的形式，但有些形式已超出本固件库的范围。为了充分利用 PWM 模块的特性，建议读者使用寄存器访问宏的方式实现。

PWM 固件库函数的详细介绍请参考书后附录 J。



12.3 例程

下面将以 TI 提供的 pwm_invert. c 程序为例来介绍 PWM 固件库的使用、编程及测试方法。

1. 在 EK－TM4C123GXL 板子上测试 pwm_invert. c 程序

1）PWM 例程硬件连线图如图 12-7 所示。

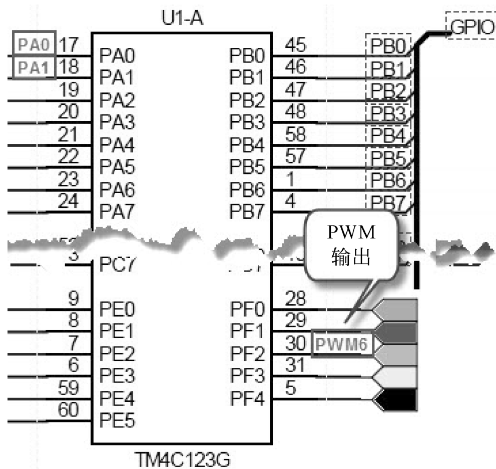


图 12-7　PWM 例程硬件连线图

2）pwm_invert. c 介绍。

```
//*****************************************************************
//!文件名：invert. c
```

```
//!来源:根据 TI 例程改编
//!功能描述:
//!例子介绍了如何设置 PWM0 使用其反向输出功能
//!此功能允许反转 PWM 的输出极性
//!该例程将一个 25% 占空比的 PWM 输出每隔 1 s 反相一次,以得到 75% 占空比的 PWM 信号
//!LaunchPad 板上 LED_B(FP2 引脚)的灯每秒闪烁 1 次来指示占空比为 25% 与 75% 之间的切换
//!例程中使用下列的外设和 I/O 信号:
//! - GPIOF 端口(用于 PWM 引脚)
//! - PWM6 - PF2
//!配置 UART 信号仅用于在 PuTTY 上显示例程中的消息
//! - UART0 外设
//! - GPIOA 端口 (用于 UART0 引脚)
//! - UART0RX - PA0
//! - UART0TX - PA1
//
// ***********************************************************************
#include < stdbool. h >
#include < stdint. h >
#include " inc/hw_memmap. h"
#include " driverlib/gpio. h"
#include " driverlib/pin_map. h"
#include " driverlib/pwm. h"
#include " driverlib/sysctl. h"
#include " driverlib/uart. h"
#include " utils/uartstdio. h"
// ***********************************************************************
//此函数设置 UART0 的控制台来显示该例程运行时的信息
// ***********************************************************************
void
InitConsole( void)
{
    //
    //使能用于 UART0 引脚功能的 GPIO 端口 A
    //
    SysCtlPeripheralEnable( SYSCTL_PERIPH_GPIOA) ;
    //
    //将引脚复用端口 A0 和 A1 配置为 UART0 功能
    //
    GPIOPinConfigure( GPIO_PA0_U0RX) ;
    GPIOPinConfigure( GPIO_PA1_U0TX) ;
    //
    //使能 UART0 以便后续配置时钟
    //
    SysCtlPeripheralEnable( SYSCTL_PERIPH_UART0) ;
    //
    //使用内部 16 MHz 振荡器作为 UART 时钟源
    //
    UARTClockSourceSet( UART0_BASE,UART_CLOCK_PIOSC) ;
    //
```

```
    //选择这些引脚的(UART)复用功能
    //
    GPIOPinTypeUART(GPIO_PORTA_BASE,GPIO_PIN_0 | GPIO_PIN_1);
    //
    //初始化 UART 的控制台 I/O
    //
    UARTStdioConfig(0,115200,16000000);
}
// ***********************************************************************
//将 PWM6 配置为在 250 Hz 下运行的占空比为 25% 的 PWM 信号,以及每隔 5 s 反相一次输出
// ***********************************************************************
int
main(void)
{
    //
    //设置时钟直接从外部晶振 16 MHz 运行
    //
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_16MHZ);
    //
    //设置 PWM 时钟
    //
    SysCtlPWMClockSet(SYSCTL_PWMDIV_1);
    //
    //设置串行控制台用于显示消息
    //
    InitConsole();
    //
    //在控制台上的显示设置
    //
    UARTprintf("    PWM -> \n");
    UARTprintf("    模块:PWM1\n");
    UARTprintf("    引脚:PF2\n");
    UARTprintf("    配置占空比:25%% \n");
    UARTprintf("    反相占空比:75%% \n");
    UARTprintf("    功能:PWM 输出每隔 1 s 反相一次 . \n\n");
    UARTprintf("    在 M1PWM6(PF2)上产生 PWM 信号 -> 状态 = ");
    //
    //使能 PWM 外设
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);
    //
    //使能 GPIOF 端口
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    //
    //将复用引脚 GPIO_PF2 配置成 M1PWM6 功能
    //
```

```
GPIOPinConfigure(GPIO_PF2_M1PWM6);
//
//将引脚 PF2 配置成 PWM 功能
//
GPIOPinTypePWM(GPIO_PORTF_BASE,GPIO_PIN_2);
//
//将 PWM6 配置为异步更新的先递增后递减计数模式
//
PWMGenConfigure(PWM1_BASE,PWM_GEN_1,PWM_GEN_MODE_UP_DOWN |
                PWM_GEN_MODE_NO_SYNC);
//
//设置 PWM 周期为 250 Hz
//计数公式为 N = (1/f) SysClk
//其中,N 为函数参数,f 为所需的频率,SysClk 为系统时钟频率
//当 f = 250 Hz 时,N = (1/250Hz) ×16MHz = 64000(时钟周期数)
//注意,最大周期应设置为 2^16
//
PWMGenPeriodSet(PWM1_BASE,PWM_GEN_0,64000);
//
//设置 PWM6 输出的占空比为 25%
//
PWMPulseWidthSet(PWM1_BASE,PWM_OUT_6,
PWMGenPeriodGet(PWM1_BASE,PWM_OUT_6)/4);
//
//使能 PWM6 位 6(PF2)输出
//
PWMOutputState(PWM1_BASE,PWM_OUT_6_BIT,true);
//
//使能 PWM 发生器模块
//
PWMGenEnable(PWM1_BASE,PWM_GEN_1);
//
//无限循环等待产生 PWM 信号
//
while(1)
{
    //
    //显示 PWM 正常输出
    //
    UARTprintf("Normal    \b\b\b\b\b\b\b\b");
    //
    //延时 1 s
    //计数公式:延时(时钟周期数) = 3 × 参数
    //
    SysCtlDelay((SysCtlClockGet() * 1)/3);
    //
    //M1PWM6 信号反相
    //
    PWMOutputInvert(PWM1_BASE,PWM_OUT_6_BIT,true);
```

```
            //
            //显示反相的 PWM 信号
            //
            UARTprintf("Inverted\b\b\b\b\b\b\b\b");
            //
            //延时 1 s
            //计数公式:延时(时钟周期数)=3×参数
            //
            SysCtlDelay((SysCtlClockGet() * 1)/3);
            //
            //切换 M1PWM6 信号返回正常操作
            //
            PWMOutputInvert(PWM1_BASE,PWM_OUT_6_BIT,false);
        }
    }
```

3）创建 pwm_invert 工程，并在 C:\ti\TivaWare_C_Series-2.0.1.11577\examples\/peripherals\pwm 目录下导入 invert.c 文件到工程中，如图 12-8 所示。

4）编译 pwm_ invert 工程并将其下载到 EK-TM4C123GXL 开发板中。

5）测试。

① 打开 PuTTY，其显示的信息如图 12-9 所示。



图 12-8 创建的 pwm_invert 工程

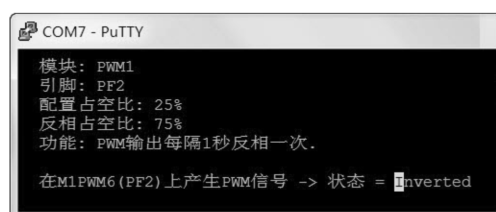


图 12-9 在 PuTTY 上显示的程序测试结果信息

从图 12-9 中可以看到，每隔 1 s PWM 输出的占空比反相一次符合对程序的设计要求。

② 在 EK-TM4C123GXL 板上也观察到了蓝色 LED 指示灯每秒闪烁一次，进一步验证了程序功能的正确性，如图 12-10 所示。

说明：有兴趣的读者可以用示波器观察 FP2 端口输出的 PWM 波是否满足要求的占空比，以及它们是否每秒反相一次。没有示波器的读者也可以用 CCS 软件画波形图的功能来查看 PWM 的输出是否满足设计要求。或者采用 Keil ARM 软件的逻辑分析仪来分析 PWM 输出的结果，当然这必须在 Keil ARM 中新建一个 pmw_invert 工程。

2. 在 Proteus 上测试 pwm_invert.c 程序

1）搭建 pwm_invert.c 程序的测试电路，如图 12-11 所示。

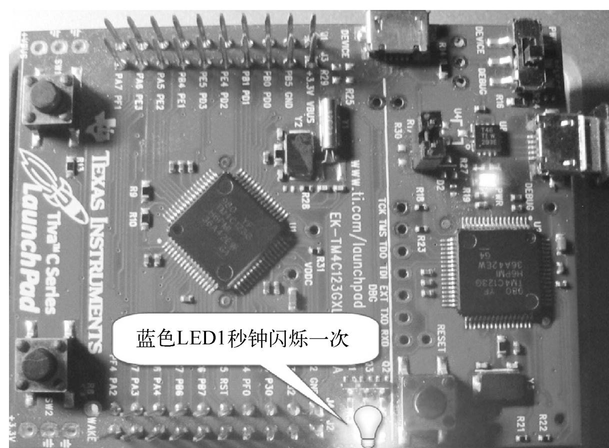


图 12-10　在 EK－TM4C123GXL 板上观察到的程序测试结果

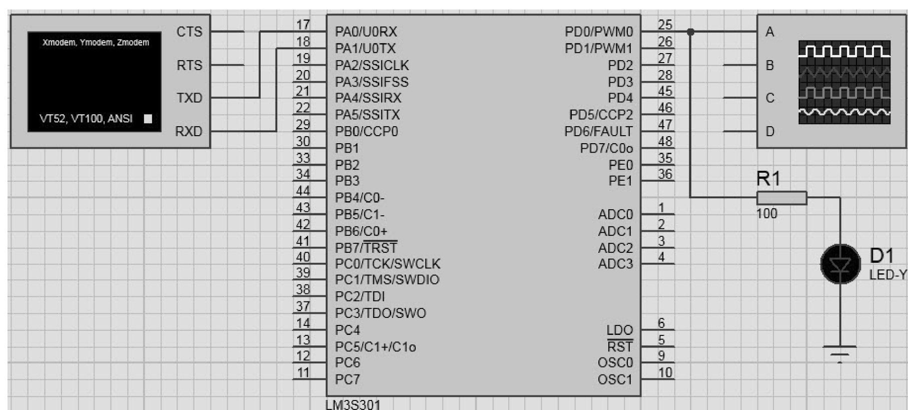


图 12-11　pwm_invert. c 程序的测试电路

2）pwm_invert. c 程序。由于这段程序和上面所介绍的 pwm_invert. c 程序基本相同，不再给出，需要注意如下：

```
//
//设置 PWM 周期为 250 Hz
//计数公式为:N = (1/f) × SysClk
//其中,N 为函数参数,f 为所需的频率,SysClk 为系统时钟频率
//当 f = 250 Hz 时,N = (1/250Hz) ×6 MHz = 24000(时钟周期数)
//注意,最大周期应设置为 2^16
//
PWMGenPeriodSet(PWM_BASE,PWM_GEN_0,24000);
```

3）pwm_invert. c 程序的测试结果如图 12-12、图 12-13 所示。

从图 12-12 中可以看到，脉冲为整个周期的 1/4，通过调节示波器旋钮让波形准确地对准显示器中的网格，就可清晰地看到这个结果，从而验证程序的正确性。同时 LED 闪烁较快，这是因为 LED 的导通时间相对较短，因此感觉 LED 闪烁较快。

从图 12-13 中可以看到，脉冲为整个周期的 3/4，通过调节示波器旋钮让波形准确地对

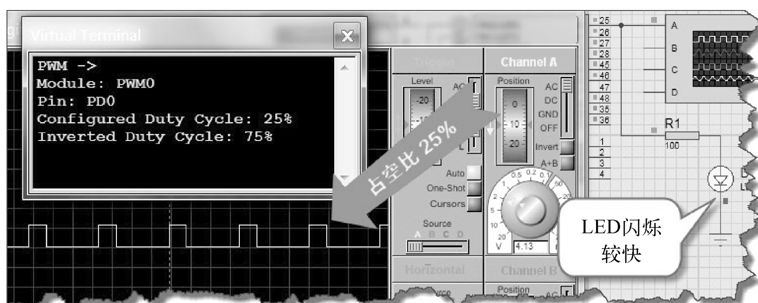


图 12-12　占空比为 25% 时的测试结果

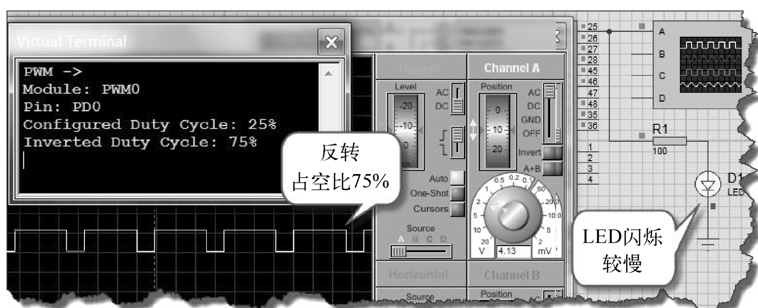


图 12-13　占空比为 75% 时的测试结果

准显示器中的网格，就可清晰地看到这个结果，从而验证程序的正确性。同时 LED 闪烁较慢，这是因为 LED 的导通时间相对较长，因此感觉 LED 闪烁较慢。

对于那些没有板子或有板子但资源较少的读者，借助 Proteus 提供的大量虚拟工具，在学习编程或项目开发初期不失为一种有效的方法。

本小节程序见配套资源第 12 章内容，如果 Pruteus 电路不能运行，要先行加载 pwm_invert. elf 文件。

第 13 章

微直接存储器访问（μDMA）

直接存储器访问（Direct Memory Access，DMA），用于实现设备与存储器之间，以及存储器与输入/输出端口之间的高速传输，而无需微控制器的干预。控制器首先必须向微控制器发出请求来获取系统的总线控制器，当完成传输之后返还系统的总线控制权。DMATM4C123GH6PM 微控制器包含一个直接存储器访问控制器，被称为微 DMA（μDMA）。它提供的工作方式可分流 Cortex－TM4C 处理器的数据传输任务，从而可更有效地利用处理器和可用的总线带宽。μDMA 控制器能自动执行存储器与外设之间的传输。每个支持片上 μDMA 模块的外设都有其专用的 μDMA 通道，通过合理的编程配置可将其设置为自动传输方式，以便在外设与存储器之间高效的传输数据。

μDMA 的 API 提供了 Tiva C uDMA 控制器的配置函数，用于与 ARM Cortex－TM4C 处理器一起工作，并为系统提供高效与低开销的数据传输模块。

该驱动程序包含在 driverlib/udma.c 中，driverlib/udma.h 包含应用程序使用的 API 定义。

本章的主要内容：

- μDMA 模块
- μDMA 宏和库函数（见书后附录 K）
- 例程



13.1 μDMA 单元

13.1.1 μDMA 的特点

μDMA 控制器具有如下特点：

1）ARM PrimeCell 32 通道可配置的 μDMA 控制器。

2）支持存储器到存储器、存储器到外设、外设到存储器的多种传输模式：

① 用于简单传输场景的基本模式。

② 用于连续数据流的乒乓模式。

③ 按照可编程的任务列表，凭借单个请求触发的多达 256 个任意传输的散集模式。

3） 高度灵活和可配置的通道操作：

① 可单独配置和独立操作的通道。

② 支持片上模块的专用通道。

③ 灵活的通道分配。
④ 给双向模块的每个接收和发送提供单独的通道。
⑤ 可软件启动传输的专用通道。
⑥ 每通道都可配置优先级方案。
⑦ 可选软件启动的任意通道请求。
4）两级优先级。
5）采用设优化计改善了 μDMA 控制器与处理器内核之间的总线访问性能：
① μDMA 控制器访问是从属于内核的访问。
② RAM 分段。
③ 外设总线分段。
6）数据宽度为 8 位、16 位或 32 位。
7）传输数据的大小为可编程的二进制步，即 2 的整数幂，其范围为 1～1024。
8）源地址和目的地址增量大小为字节、半字、字或没有增量。
9）可屏蔽的外设请求。
10）传输完成后将发生中断，每通道都有独立的中断。

13.1.2 μDMA 模块框图

μDMA 模块框图如图 13-1 所示。

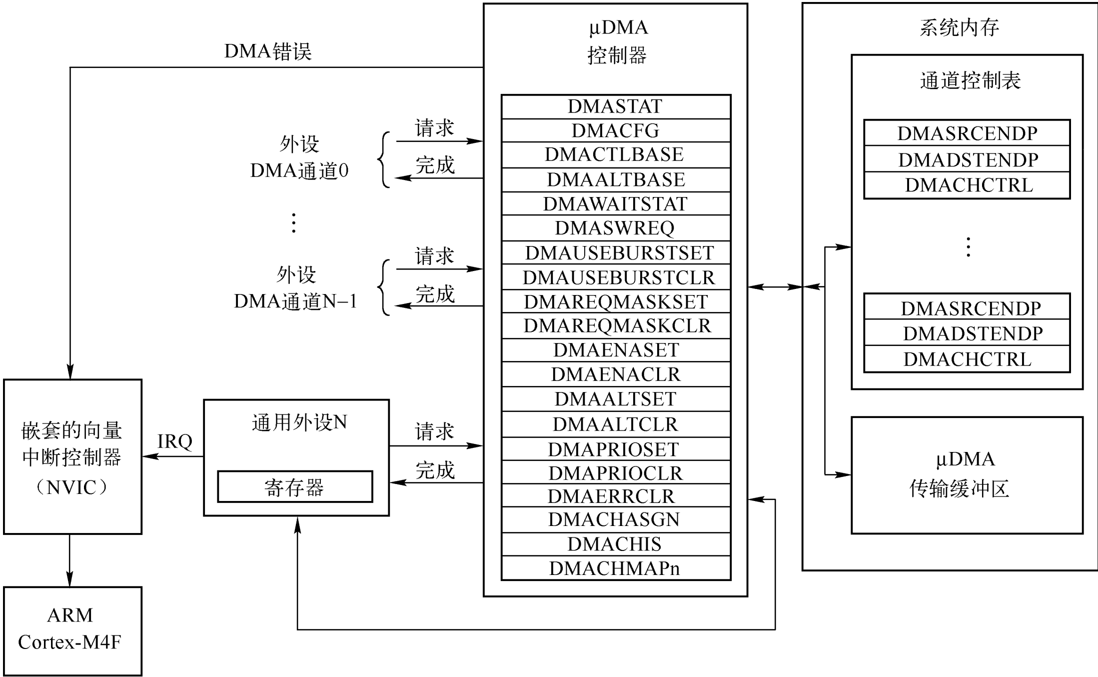


图 13-1　μDMA 模块框图

13.1.3 功能简介

μDMA 控制器是一种灵活和高可配置的 DMA 控制器，被设计成能配合 Cortex－TM4C 处

理器的内核工作以实现高效的数据传输。它支持多种数据宽度与地址递增策略，且各 DMA 通道之间具有不同的优先级和多种传输模式，允许复杂的可编程数据传输。μDMA 控制器占用总线总是从属于处理器内核，所以它不会影响到处理器的总线事务。因为 μDMA 控制器只占用处理器内核的空闲总线周期，它提供的数据传输带宽基本上是不受约束的，因而不影响系统的其他部分。通过对总线架构的优化大大提升了处理器内核和 μDMA 控制器有效地共享片上总线的能力，从而提高了它的性能。优化包括 RAM 分段和外设总线分段，在许多情况下，允许处理器内核心和 μDMA 控制器同时访问总线并进行数据传输。

μDMA 控制器可以和从片上 SRAM 进行双向数据传输。然而，由于闪存和 ROM 都位于一个单独的内部总线，因此闪存或 ROM 不可能传输数据到 μDMA 控制器中。

每个 μDMA 控制器支持的外设功能，都有一个可被独立配置的专用通道。μDMA 控制器使用在系统存储器中由处理器维护的通道控制结构体，实现了一种独特的配置方法。不但支持简单的传输模式，而且它也可以在存储器中建立复杂的“任务”列表，允许 μDMA 控制器收到单次传输请求后，根据“任务列表”执行任意大小的数据传输和接收从任意位置传送过来的数据。μDMA 控制器还支持使用乒乓缓冲以适应与外设的双向数据流。

每个通道还可配置其仲裁大小，是指在 μDMA 控制器重新仲裁通道的优先级之前，以突发方式传送的项目数量。当每次遇到一个 μDMA 服务请求时，根据配置的仲裁数目，可精准控制到底有多少项目与外设之间发生了数据传输。

1. 通道分配

μDMA 控制器通道分配见表 13-1。

表 13-1　μDMA 通道分配

Enc.	0		1		2		3		4	
Ch#	外　设	类型	外　设	类型	外　设	类型	外　设	类型	外　设	类型
0	USB0 EP1 RX	SB	UART2 RX	SB	软件	B	GPTimer 4A	B	软件	B
1	USB0 EP1 TX	B	UART2 TX	SB	软件	B	GPTimer 4B	B	软件	B
2	USB0 EP2 RX	B	GPTimer 3A	B	软件	B	软件	B	软件	B
3	USB0 EP2 TX	B	GPTimer 3B	B	软件	B	软件	B	软件	B
4	USB0 EP3 RX	B	GPTimer 2A	B	软件	B	GPIO A	B	软件	B
5	USB0 EP3 TX	B	GPTimer 2B	B	软件	B	GPIO B	B	软件	B
6	软件	B	GPTimer 2A	B	UART5 RX	SB	GPIO C	B	软件	B
7	软件	B	GPTimer 2B	B	UART5 TX	SB	GPIO D	B	软件	B
8	UART0 RX	SB	UART1 RX	SB	软件	B	GPTimer 5A	B	软件	B
9	UART0 TX	SB	UART1 TX	SB	软件	B	GPTimer 5B	B	软件	B
10	SSI0 RX	SB	SSI1 RX	SB	UART6 RX	SB	GPWideTimer 0A	B	软件	B
11	SSI0 TX	SB	SSI1 TX	SB	UART6 TX	SB	GPWideTimer 0B	B	软件	B
12	软件	B	UART2 RX	SB	SSI2 RX	SB	GPWideTimer 1A	B	软件	B
13	软件	B	UART2 TX	SB	SSI2 TX	SB	GPWideTimer 1B	B	软件	B
14	ADC0 SS0	B	GPTimer 2A	B	SSI3 RX	SB	GPIO E	B	软件	B
15	ADC0 SS1	B	GPTimer 2B	B	SSI3 TX	SB	GPIO F	B	软件	B
16	ADC0 SS2	B	软件	B	UART3 RX	SB	GPWideTimer 2A	B	软件	B
17	ADC0 SS3	B	软件	B	UART3 TX	SB	GPWideTimer 2B	B	软件	B

（续）

Enc.	0		1		2		3		4	
Ch#	外　设	类型	外　设	类型	外　设	类型	外　设	类型	外　设	类型
18	GPTimer 0A	B	GPTimer 1A	B	UART4 RX	SB	GPIO B	B	软件	B
19	GPTimer 0B	B	GPTimer 1B	B	UART4 TX	SB	软件	B	软件	B
20	GPTimer 1A	B	软件	B	UART7 RX	SB	软件	B	软件	B
21	GPTimer 1B	B	软件	B	UART7 TX	SB	软件	B	软件	B
22	UART1 RX	SB	软件	B	软件	B	软件	B	软件	B
23	UART1 TX	SB	软件	B	软件	B	软件	B	软件	B
24	SSI1 RX	SB	ADC1 SS0	B	软件	B	GPWideTimer 3A	B	软件	B
25	SSI1 TX	SB	ADC1 SS1	B	软件	B	GPWideTimer 3B	B	软件	B
26	软件	B	ADC1 SS2	B	软件	B	GPWideTimer 4A	B	软件	B
27	软件	B	ADC1 SS3	B	软件	B	GPWideTimer 4B	B	软件	B
28	软件	B	软件	B	软件	B	GPWideTimer 5A	B	软件	B
29	软件	B	软件	B	软件	B	GPWideTimer 5B	B	软件	B
30	软件	B	软件	B	软件	B	软件	B	软件	B
31	保留	B	保留	B	保留	B	保留	B	保留	B

注：1. S 为单个请求、B 为突发请求、SB 为单个加突发请求。

2. 表中注明“软件”的通道可用于将来外设的扩展，这些通道目前仅软件可访问，不可连接外设。30 号通道为软件专用通道。

2. 优先级

每个通道 μDMA 的优先级由通道的序号和通道的优先级标志位决定。第 0 号 μDMA 通道的优先级最高，通道的序号越大，反而其优先级越低。每个 μDMA 通道都有一个可设置的优先级标志位来确定其优先级高低。若某个通道的优先级位置位，则该通道将具有高优先级，其优先于所有未将此标志位置位的通道。假如有多个通道都设为高优先级，这时将按通道序号来区分它们的优先级高低。通道的优先级位可通过 DMA 通道优先置位寄存器（DMAPRIOSET）置位，而通过 DMA 通道优先清除寄存器（DMAPRIOCLR）清零。

3. 仲裁个数

当某个 μDMA 通道请求传输时，μDMA 控制器将对所有发出请求的通道进行仲裁，并且只向优先级最高的通道提供服务。一旦传输开始，首先将持续传输一定数量的数据之后再对发出请求的通道进行仲裁。每个通道的仲裁个数都是可设置的，其有效范围为 1 ~ 1024 个项目。当 μDMA 控制器按照仲裁个数传输了若干个项目之后，随后将检查所有发出请求的通道，并只向优先级最高的通道提供服务。

若某个优先级较低的 μDMA 通道仲裁个数设置过大，可能使高优先级通道的传输延迟增加，这是因为，μDMA 控制器需要等待低优先级的突发传输完全结束之后才会重新进行仲裁，以检查是否存在更高优先级的请求。因此，建议对于低优先级通道的仲裁个数不应设置过大，以充分保障系统对高优先级 μDMA 通道的响应速度。

4. 请求类型

μDMA 控制器可用于响应来自外设的两种请求：单次请求或突发请求。每种外设可支持其中一种或两种类型，单次请求表示外设已准备好传输一个项目，而突发请求则表示外设已

准备好传输多个项目。

根据外设发出的是单次请求还是突发请求，μDMA 控制器的响应也会有所不同。假设同时发出了单次请求和突发请求，且 μDMA 通道已按突发请求设置，则会优先响应突发请求。各种外设对这两种请求类型的支持见表 13-2。

表 13-2　支持的请求类型

外　　设	产生单个请求的事件	产生突发请求的事件
ADC	无	FIFO 半满
通用定时器	无	触发事件
GPIO	原始中断脉冲	无
SSI TX	TX FIFO 不满	TX FIFO 深度（固定 4 级）
SSI RX	RX FIFO 非空	RXFIFO 深度（固定 4 级）
UART TX	TX FIFO 不满	TXFIFO 深度（可配置）
UART RX	RX FIFO 非空	RXFIFO 深度（可配置）
USB TX	无	FIFO TXRDY
USB RX	无	FIFO RXRDY

5. 通道配置

μDMA 控制器采用在系统内存中保存一个控制表，该表包含多个通道控制的结构体，而每个 μDMA 通道在控制表中可能有一个或两个结构体。该表的每个结构体都包含源指针、目的指针、传输个数、传输模式。该表可定义到系统内存中的任意连续位置，并按 1024 字节边界对齐。

表 13-3 列出了通道控制表的内存布局。结构体由主控制结构体和副控制结构体组成。在控制表中，所有主控制结构体都在表的前半部分，而副控制结构体都在表的下半部分。在较简单的传输模式中，对传输的连续性要求不高，允许在每次传输结束后再重新配置、重新启动。这种情况一般不需要副控制结构体，因此内存中只需放置表的前半部分即可，而后半部分所占用的存储空间可另作它用。如果采用加复杂的传输模式（如乒乓模式或散聚模式），就需要用到副控制结构体，此时整个控制表都必须加载到存储器中。

对于控制表中任何未用到的存储块都可留给应用程序使用，包括任何应用程序未用的通道控制结构体与各个通道中未用到的控制字。

表 13-3　控制结构体存储器映射

偏　移　量	通　　道
0x0	0，主功能
0x10	1，主功能
...	...
0x1F0	31，主功能
0x200	0，副功能
0x210	1，副功能
...	...
0x3F0	31，副功能

表 13-4 列出了控制表中单个控制结构体项的内容，并按照 16 字节边界对齐方式。每个结构体项由 4 个长整型项组成：源末端指针、目的末端指针、控制字与一个未用的长整型项。末端指针是指向传输过程最末一个单元地址的指针（包含其本身）。若源地址或目的地址无需自动递增（如外设寄存器），则指针应当指向待传输的地址。

表 13-4　通道控制结构体

偏　移　量	描　　述
0x000	源末端指针
0x004	目标末端指针
0x008	控制字
0x00C	未使用

μDMA 控制器在传输进行时会自动更新传输大小字段和传输模式字段。在传输完成之后，会使传输个数变为 0，传输模式变为“已停止”。因控制字是由 μDMA 控制器自动修改的，则在每次新建传输之前必须手动配置，而源末端指针和目的末端指针不会被自动修改，只要源地址或目的地址不变，就无需再度配置。

在启动传输之前，必须先将 DMA 通道使能设置寄存器（DMAENASET）中的相应标志位置位来使能 μDMA 通道。当要禁止某个通道时，可将 DMA 通道使能清除寄存器（DMAENACLR）中的相应标志位置位即可，此外，在某个通道的 μDMA 传输完成后，控制器将自动关闭该通道。

6. 传输模式

μDMA 控制器支持多种传输模式，前两种模式支持简单的单次传输，而后几种模式可实现持续数据传输。

（1）停止模式

停止模式虽为控制字中传输模式字段的有效值，但它并不是一种真正的传输模式。当控制字中的传输模式为停止模式时，μDMA 控制器并不会对该通道进行任何传输，而一旦该通道使能时，μDMA 控制器还将自动关闭该通道。在任何 μDMA 传输结束后，μDMA 控制器都会自动将通道控制字的传输模式字段改为停止模式。

（2）基本模式

当设备发出一个有效请求时，μDMA 控制器执行简单数据传输。此模式用于在外设发出有效的请求信号后，无论何时数据都将被传输。如果请求信号无效，无论是否传输结束都将停止传输。在基本模式中，当所有项目传输结束后，μDMA 控制器会自动将该通道置为停止模式。

（3）自动请求模式

执行一个由请求启动的简单传输，即使请求被撤回，传输也将继续进行，直到完成整个数据传输。此模式适用于由软件启动的传输。

（4）乒乓模式

该模式用于两个缓冲器之间的数据传输，当填充每个缓冲区时，可从一个缓冲区切换到另一个缓冲区。虽然这种模式可确保外设之间收发连续的数据流，但需要对中断处理器中作更复杂的设置，并使用处理程序来管理这种所谓的乒乓缓冲器。一个该类型的示例如图 13-2 所示。

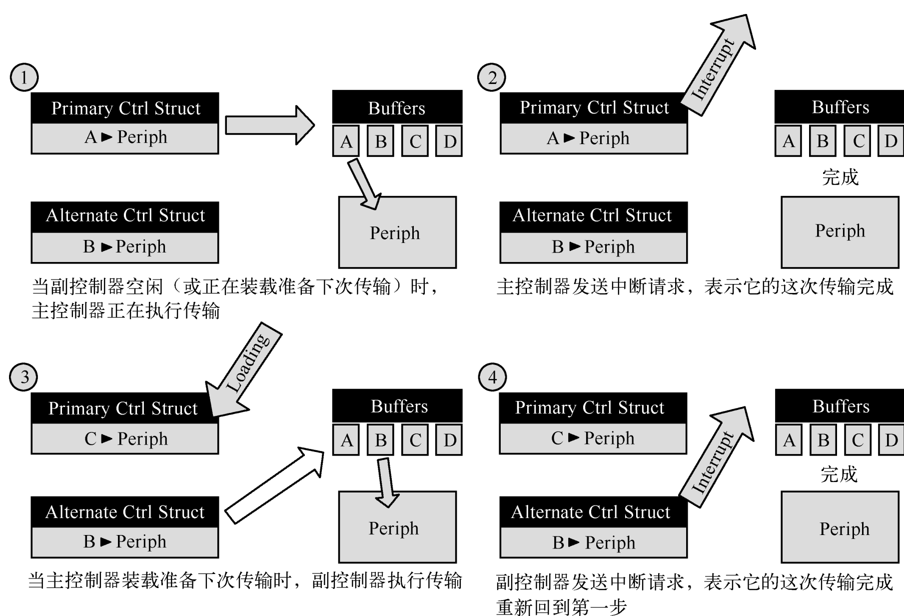


图 13-2　乒乓式 μDMA 数据会话示例

（5）存储器散聚模式

为 μDMA 控制器提供建立传输“任务”列表方法的一种复杂模式。数据块可以在存储器中的任意存储单元进行双向传输，存储器散聚模式是一种较为复杂的工作模式。通常在搬运数据块时，源数据和目标数据都是按线性分布的；然而有时必须将这些在内存中连续分布的数据分散传递到若干个不同的存储区域，或将内存中几个不同存放单元的数据块汇聚传递到同一个存储区域连续存放，此时就应当采用散聚模式。

在存储器散聚模式下，主控制结构体是按内存中一个表的内容来配置副控制结构体的。该表由处理器软件创建，包含若干个控制结构体，每个控制结构体中又包含可实现特定传输的源末端指针、目的末端指针、控制字。在每个控制结构体的控制字中必须将传输模式设置为散聚模式。主传输流程依次将表中的控制结构体复制到副控制结构体中，然后执行。μDMA 控制器就这样交替切换，即每次用主控制结构体从列表中将下一个待传输的流程配置复制到副控制结构体中，再切换到副控制结构体去执行相应的传输任务。在列表的最末一个控制结构体中，应将其控制字编程为自动传输模式。因此当执行到最后一个传输过程时为自动模式，μDMA 控制器在操作完成后将停止该通道的运行。仅当最后一次传输过程也结束时，才会发出结束中断。若将控制表最后一个控制结构体重新指向列表的起始位置（或指向一个新的列表），将使整个列表处于循环状态。

可从如图 13-3、图 13-4 所示的例子中看到存储器散聚模式的工作过程。

在图 13-4 中，使用通道的主控制结构体，μDMA 将任务 A 的配置复制到通道的副控制结构体中，然后 μDMA 再将数据从源缓冲区 A 复制到目标缓冲区中。（仅以任务 A 为例，任务 B 与 C 工作过程相同）

7. 传输个数及增量

μDMA 控制器所支持的传输数据宽度为 8 位、16 位或 32 位，对于任何传输，都必须保证源数据与目标数据的宽度相同。源地址与目标地址可以按字节、半字或字自动递增，也可

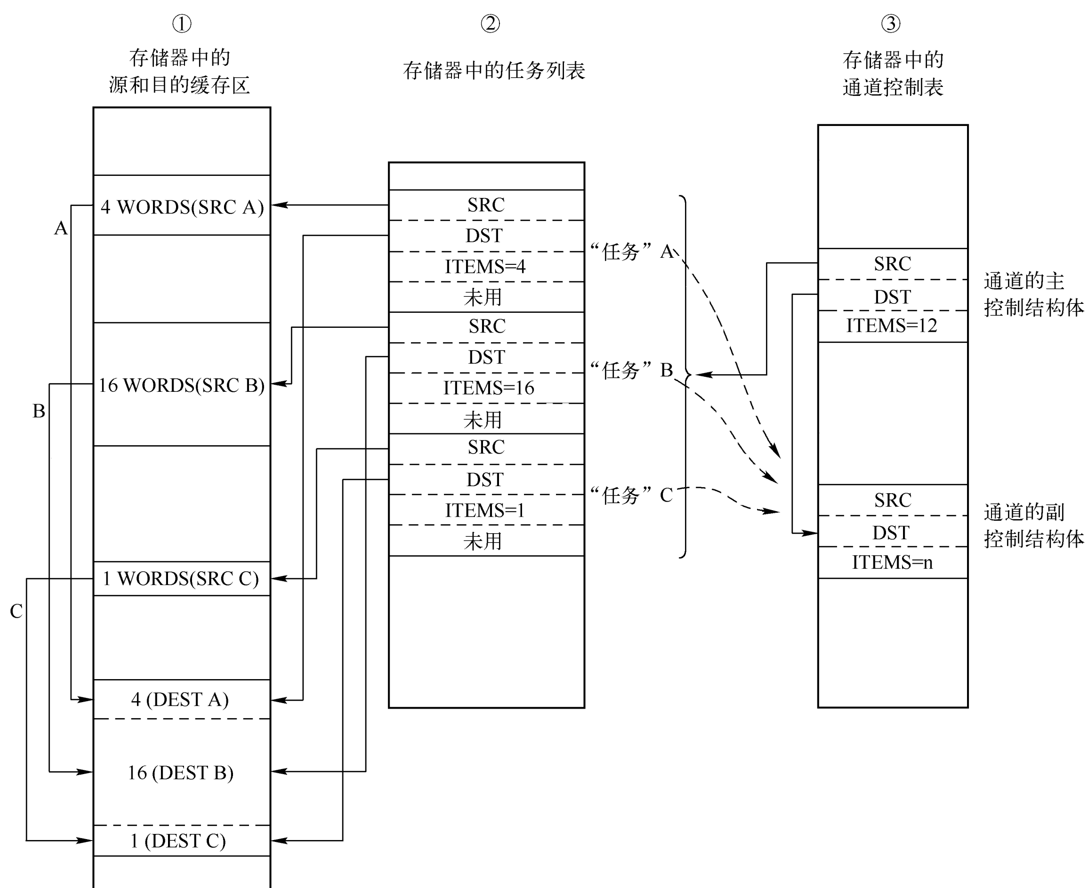


图 13-3　存储器散聚模式（创建及配置）

注：

1. 应用程序需要将存储器中三个不同存储单元的若干个项目复制到一个缓冲区中并顺序组合。
2. 应用程序在存储器中建立 μDMA "任务列表"，表中包括 3 个 μDMA 控制 "任务" 指针与控制配置。
3. 应用程序设置通道的主控制结构体，将任务逐个复制到副控制结构体中，然后由 μDMA 控制器执行。

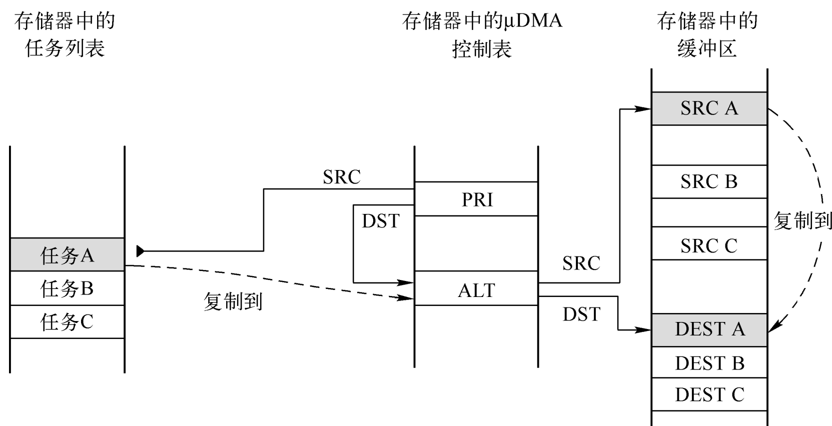


图 13-4　存储器散聚模式（μDMA 复制序列）

设置为非自动递增。源地址增量与目标地址增量无关联，其设置仅要求保证其大于等于数据宽度即可。

8. 外设接口

如果某个外设支持 μDMA 功能，则在它准备好数据传输时，可发出一个单次请求信号和/或一个突发请求信号。请求信号可通过 DMA 通道请求屏蔽置位寄存器（DMAREQMASKSET）使能，并通过 DMA 通道请求屏蔽清零位寄存器（DMAREQMASKCLR）来禁止。若某个通道的请求屏蔽位置位，则禁止该通道的 μDMA 请求信号。若 μDMA 通道已被正确配置和使能，则当外设发出请求信号时，μDMA 控制器将启动传输过程。当 μDMA 传输结束时，μDMA 控制器将会发出一个中断。注意：当使用 μDMA 与外设进行数据传输时，外设必须禁止所有 NVIC 中的中断。

9. 软件请求

在 32 个 μDMA 通道中有一个专门用于软件启动的传输通道。当该通道 μDMA 传输完成时，有专用的中断予以指示。要正确使用软件启动的 μDMA 传输，应先行配置与使能传输过程，然后通过 DMA 通道软件请求寄存器（DMASWREQ）发出软件请求。注意，基于软件的 μDMA 传输应当采用自动传输模式。

通过 DMASWREQ 寄存器也可以启动任意通道的 μDMA 传输。若在某个外设的 μDMA 通道上使用软件启动请求，在传输完成时，结束中断将在该外设的中断向量处发生，而不是软件中断向量。只要某个外设不用 μDMA 做数据传输，则任何外设通道都可用作软件传输请求。

10. 中断与错误

当某个 μDMA 传输过程结束时，μDMA 控制器将在相应外设的中断向量处发出一个结束中断。因此，若某个外设使用 μDMA 传输数据时，并且使能了该外设的中断，则在中断处理函数中必须包含对 μDMA 传输结束时的中断处理。如果传输过程使用了软件 μDMA 通道，则结束中断将在专用软件 μDMA 中断向量上产生（见表 13-5）。

表 13-5 μDMA 中断分配

中　　断	分　　配
46	μDMA 软件通道传输中断
47	μDMA 错误中断

当使能某外设的 μDMA 后，μDMA 控制器将禁止该外设的普通传输中断传递到中断控制器中，但该中断的状态仍可在外设的中断寄存器中查询到。因此，当使用 μDMA 传输大量数据时，中断控制器并不会跟随数据流从外设频繁地接收中断，而是仅在数据传输结束时才会收到一个中断。注意，未屏蔽的外设错误中断仍会发送到中断控制器中。

当 μDMA 通道发出一个完成中断时，在 DMA 通道中断状态寄存器（DMACHIS）中对应该通道的 CHIS 位将置位。外设中断处理代码也可通过该寄存器来确定中断是由 μDMA 通道产生的，还是由外设中断寄存器上报的出错事件造成的。当中断处理程序激活后，μDMA 控制器所产生的结束中断请求将会自动清除。

若 μDMA 控制器在尝试进行数据传输时遇到了总线或存储器保护错误时，将会自动关闭出错的 μDMA 通道，并在 μDMA 错误中断向量处发生中断。处理器可通过读取 DMA 总线错误清除寄存器（DMAERRCLR）来确定是否产生了错误中断。一旦产生错误则 ERRCLR

标志位将置位，可向 ERRCLR 位写 1 来清除错误状态。



13.2 μDMA 固件库函数

μDMA API 提供了一组用于配置 Tiva μDMA 控制器高效执行 DMA 传输的函数。设置和执行 μDMA 传输，一般的函数调用顺序如下：

① μDMAEnable( )。被调用一次来使能控制器。

② μDMAControlBaseSet( )。被调用一次来设置通道控制表。

③ μDMAChannelAttributeEnable( )。被调用一次或不经常配置通道的行为。

④ μDMAChannelControlSet( )。用于设置数据传输的特性。如果数据传输的特性不变，仅需调用该函数一次。

⑤ μDMAChannelTransferSet( )。用于设置传输的缓冲区指针和大小。需在新的传输开始前调用此函数。

⑥ μDMAChannelEnable( )。使能一个通道用于执行数据传输。

⑦ μDMAChannelRequest( )。用于启动一个基于软件的传输。此函数通常不用在基于外设的传输中。

注意：控制表必须在 1024 字节边界对齐。

μDMA 控制器的固件库函数由两部分组成：定义文档和函数文档。

有关 μDMA 固件库较详细的函数说明见书后附录 K。



13.3 例程

本小节将以 TI 提供的例程 udma_demo 为例来介绍 μDMA 固件库的使用及编程方法。

1） udma_demo. c 程序说明。

```
// ***********************************************************************
//!该例程介绍了使用 μDMA 控制器在存储器之间传输数据,以及把数据传输到 μDMA 或从其接
//!收数据。测试运行 10 s 后推出
// ***********************************************************************
//包含文件
…. (省略,见配套资源第 13 章程序)
// ***********************************************************************
//SysTick 中断每秒使用 SysTick 的个数
// ***********************************************************************
#define SYSTICKS_PER_SECOND          100
// ***********************************************************************
//存储器传输的源和目标缓冲区的大小(字)
// ***********************************************************************
#define MEM_BUFFER_SIZE              1024
// ***********************************************************************
//UART 发送和接收缓冲区的大小
// ***********************************************************************
#define UART_TXBUF_SIZE              256
```

```
#define UART_RXBUF_SIZE                256
// ****************************************************************************
//用于存储器传输的源和目标缓冲区
// ****************************************************************************
static uint32_t g_ui32SrcBuf[MEM_BUFFER_SIZE];
static uint32_t g_ui32DstBuf[MEM_BUFFER_SIZE];
// ****************************************************************************
//用于 UART 传输的发送和接收缓冲器。这里使用有一个发送缓冲区和一对接收乒乓缓冲区
// ****************************************************************************
static uint8_t g_ui8TxBuf[UART_TXBUF_SIZE];
static uint8_t g_ui8RxBufA[UART_RXBUF_SIZE];
static uint8_t g_ui8RxBufB[UART_RXBUF_SIZE];
// ****************************************************************************
//μDMA 的错误计数,其值由 μDMA 错误处理器递增
// ****************************************************************************
static uint32_t g_ui32uDMAErrCount = 0;
// ****************************************************************************
//μDMA 中断发生的次数,但 μDMA 传输并未完成。该变量应该保持为 0
// ****************************************************************************
static uint32_t g_ui32BadISR = 0;
// ****************************************************************************
//每个乒乓缓冲区需在 UART 缓冲区中填充的个数
The count of UART buffers filled,one for each ping - pong buffer.
// ****************************************************************************
static uint32_t g_ui32RxBufACount = 0;
static uint32_t g_ui32RxBufBCount = 0;
// ****************************************************************************
//μDMA 传输存储器块计数。该值每当一个存储器块传送完成将由 μDMA 中断处理器负责递增
// ****************************************************************************
static uint32_t g_ui32MemXferCount = 0;
// ****************************************************************************
//16.16 定点格式的 CPU 使用率百分比
// ****************************************************************************
static uint32_t g_ui32CPUUsage;
// ****************************************************************************
//程序开始所花费的秒数,这个值由 SysTick 中断处理程序维护
// ****************************************************************************
static uint32_t g_ui32Seconds = 0;
// ****************************************************************************
//所使用的 μDMA 控制器的控制表,此表必须在 1024 字节边界对齐
// ****************************************************************************
#if          defined(ewarm)
#pragma      data_alignment = 1024
uint8_t      ui8ControlTable[1024];
#elif        defined(ccs)
#pragma      DATA_ALIGN(ui8ControlTable,1024)
uint8_t      ui8ControlTable[1024];
#else
uint8_t      ui8ControlTable[1024] __attribute__ ((aligned(1024)));
```

```
#endif
….
void
SysTickHandler(void)
{
    static uint32_t ui32TickCount = 0;
    //递增节拍计数器
    ui32TickCount ++;
    //如果发生每秒的节拍数,则递增秒计数器
    if(!(ui32TickCount % SYSTICKS_PER_SECOND))
    {
        g_ui32Seconds ++;
    }
    //调用 CPU 使用率滴答函数
    g_ui32CPUUsage = CPUUsageTick();
}
// ***********************************************************************
//μDMA 错误的中断处理程序
// ***********************************************************************
void
uDMAErrorHandler(void)
{
    uint32_t ui32Status;
    //检查 μDMA 错误位
    ui32Status = ROM_uDMAErrorStatusGet();
    //如果有一个 μDMA 错误,则清除错误并将错误计数器加 1
    if(ui32Status)
    {
        ROM_uDMAErrorStatusClear();
        g_ui32uDMAErrCount ++;
    }
}
// ***********************************************************************
//存储器通道的 μDMA 中断处理程序。这个中断将使计数器加 1,然后重新启动另一个存储器传输
// ***********************************************************************
void
uDMAIntHandler(void)
{
    uint32_t ui32Mode;
    //检查主控制结构来指示完成
    ui32Mode = ROM_uDMAChannelModeGet(UDMA_CHANNEL_SW);
    if(ui32Mode == UDMA_MODE_STOP)
    {
        //传输完成计数值加 1
        g_ui32MemXferCount ++;
        //配置另一个传输
        ROM_uDMAChannelTransferSet(UDMA_CHANNEL_SW,UDMA_MODE_AUTO,
                                   g_ui32SrcBuf,g_ui32DstBuf,
                                   MEM_BUFFER_SIZE);
```

```
            //启动另一个传输
            ROM_uDMAChannelEnable(UDMA_CHANNEL_SW);
            ROM_uDMAChannelRequest(UDMA_CHANNEL_SW);
        }
        //如果通道不停止,则存在错误
        else
        {
            g_ui32BadISR ++;
        }
}
// ************************************************************************
//UART1 中断处理程序
// ************************************************************************
void
UART1IntHandler(void)
{
        uint32_t ui32Status;
        uint32_t ui32Mode;
        //读 UART 的中断状态
        ui32Status = ROM_UARTIntStatus(UART1_BASE,1);
        //清除任何挂起状态
        ROM_UARTIntClear(UART1_BASE,ui32Status);
        //检查 DMA 控制表,以查看乒乓"A"传输是否完成。"A"传输使用了接收缓冲区"A",及主
        //控制结构
        ui32Mode = ROM_uDMAChannelModeGet(UDMA_CHANNEL_UART1RX | UDMA_PRI_SELECT);
        //如果主控制结构指示停止,则"A"接收缓冲区完成。μDMA 控制器将进入"B"的缓冲区继
        //续接收数据
        if(ui32Mode == UDMA_MODE_STOP)
        {
            //计数器加 1 来指示数据已接收到缓冲区 A 中
            g_ui32RxBufACount ++;
            //使用主控制结构给 "A"的缓冲区设置下一次传输
            ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1RX | UDMA_PRI_SELECT,
                                       UDMA_MODE_PINGPONG,
                                       (void *)(UART1_BASE + UART_O_DR),
                                       g_ui8RxBufA,sizeof(g_ui8RxBufA));
        }
        //检查 DMA 控制表,以查看乒乓"B"传输是否完成。"B"传输使用了接收缓冲区"B",及备
        //用控制结构
        ui32Mode = ROM_uDMAChannelModeGet(UDMA_CHANNEL_UART1RX | UDMA_ALT_SELECT);
        //如果备用控制结构指示停止,则"B"接收缓冲区完成。μDMA 控制器将进入"A"的缓冲区
        //继续接收数据
        if(ui32Mode == UDMA_MODE_STOP)
        {
            //计数器加 1 来指示数据已接收到缓冲区 A 中
            g_ui32RxBufBCount ++;
            //使用备用控制结构给"B"的缓冲区设置下一次传输
            ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1RX | UDMA_ALT_SELECT,
```

```
                                                    UDMA_MODE_PINGPONG,
                                                    (void *)(UART1_BASE + UART_O_DR),
                                                    g_ui8RxBufB,sizeof(g_ui8RxBufB));
    }
    //如果 UART1 TX DMA 通道被禁止,则 TX DMA 传输完成
    if(!ROM_uDMAChannelIsEnabled(UDMA_CHANNEL_UART1TX))
    {
        //启动另一个到 UART1 TX 的 DMA 传输
        ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1TX | UDMA_PRI_SELECT,
                                                    UDMA_MODE_BASIC,g_ui8TxBuf,
                                                    (void *)(UART1_BASE + UART_O_DR),
        sizeof(g_ui8TxBuf));
        //uDMA TX 通道必须被重新使能
        ROM_uDMAChannelEnable(UDMA_CHANNEL_UART1TX);
    }
}
// ***********************************************************************
//初始化 UART1 外设并设置 TX 和 RX μDMA 通道。UART 配置为回环模式,使发送的任何数据
//将在 RX 接收。μDMA 通道被配置成在 TX 通道将缓冲区中的数据复制到 UART TX 输出。而
//μDMA RX 通道将接收到的任何输入数据存放到一对乒乓模式的缓冲区中
// ***********************************************************************
void
InitUART1Transfer(void)
{
    unsigned int uIdx;
    //用简单的数据模式填充 TX 缓冲区
    for(uIdx = 0; uIdx < UART_TXBUF_SIZE; uIdx ++ )
    {
        g_ui8TxBuf[uIdx] = uIdx;
    }
    //启用 UART 外设,并将其配置为运行即使 CPU 处于睡眠
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_UART1);
    //配置 UART 的通信参数
    ROM_UARTConfigSetExpClk(UART1_BASE,ROM_SysCtlClockGet(),115200,
                                        UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                                        UART_CONFIG_PAR_NONE);
    //设置 TX 和 RX 的 FIFO 深度都为 4
    ROM_UARTFIFOLevelSet(UART1_BASE,UART_FIFO_TX4_8,UART_FIFO_RX4_8);
    //使能 UART 操作,并使能 TX 和 RX 通道的 μDMA 接口
    ROM_UARTEnable(UART1_BASE);
    ROM_UARTDMAEnable(UART1_BASE,UART_DMA_RX | UART_DMA_TX);
    //寄存器写将设置 UART 在回环模式下运行。发送到 TX 输出的任何数据将在 RX 输入接收
    HWREG(UART1_BASE + UART_O_CTL) | = UART_CTL_LBE;
    //使能 UART 外设中断
    ROM_IntEnable(INT_UART1);
    //禁止 μDMA UART1RX 通道的属性设置
    ROM_uDMAChannelAttributeDisable(UDMA_CHANNEL_UART1RX,
```

```
                                    UDMA_ATTR_ALTSELECT | UDMA_ATTR_USEBURST |
                                    UDMA_ATTR_HIGH_PRIORITY |
                                    UDMA_ATTR_REQMASK);
    //在 UART RX 通道给主控制结构配置控制参数
    ROM_uDMAChannelControlSet(UDMA_CHANNEL_UART1RX | UDMA_PRI_SELECT,
                              UDMA_SIZE_8 | UDMA_SRC_INC_NONE | UDMA_DST_INC_8 |
                              UDMA_ARB_4);
    //在 UART RX 通道给备用控制结构配置控制参数
    ROM_uDMAChannelControlSet(UDMA_CHANNEL_UART1RX | UDMA_ALT_SELECT,
                              UDMA_SIZE_8 | UDMA_SRC_INC_NONE | UDMA_DST_INC_8 |
                              UDMA_ARB_4);
    //设置 UART RX 主控制结构中的传输参数
    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1RX | UDMA_PRI_SELECT,
                               UDMA_MODE_PINGPONG,
                               (void *)(UART1_BASE + UART_O_DR),
                               g_ui8RxBufA, sizeof(g_ui8RxBufA));
    //设置 UART RX 备用控制结构中的传输参数
    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1RX | UDMA_ALT_SELECT,
                               UDMA_MODE_PINGPONG,
                               (void *)(UART1_BASE + UART_O_DR),
                               g_ui8RxBufB, sizeof(g_ui8RxBufB));
    //禁止 μDMA UART1TX 通道的属性设置
    ROM_uDMAChannelAttributeDisable(UDMA_CHANNEL_UART1TX,
                                    UDMA_ATTR_ALTSELECT |
                                    UDMA_ATTR_HIGH_PRIORITY |
                                    UDMA_ATTR_REQMASK);
    //使能 μDMA UART1TX 通道的属性设置
    ROM_uDMAChannelAttributeEnable(UDMA_CHANNEL_UART1TX, UDMA_ATTR_USEBURST);
    //配置 UART TX 的控制参数
    ROM_uDMAChannelControlSet(UDMA_CHANNEL_UART1TX | UDMA_PRI_SELECT,
                              UDMA_SIZE_8 | UDMA_SRC_INC_8 | UDMA_DST_INC_NONE |
                              UDMA_ARB_4);
    //设置为 μDMA UART TX 通道的传输参数
    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1TX | UDMA_PRI_SELECT,
                               UDMA_MODE_BASIC, g_ui8TxBuf,
                               (void *)(UART1_BASE + UART_O_DR),
    sizeof(g_ui8TxBuf));
    //使能 μDMA UART TX 和 RX 通道,此时 μDMA UART TX 和 RX 通道都准备开始传输
    ROM_uDMAChannelEnable(UDMA_CHANNEL_UART1RX);
    ROM_uDMAChannelEnable(UDMA_CHANNEL_UART1TX);
}
// ****************************************************************************
//初始化 μDMA 软件通道以执行存储器到存储器的 μDMA 传输
// ****************************************************************************
void
InitSWTransfer(void)
{
```

```
    unsigned int uIdx;
    //用简单的递增模式填充源存储器缓冲区
    for(uIdx = 0; uIdx < MEM_BUFFER_SIZE; uIdx ++ )
    {
        g_ui32SrcBuf[uIdx] = uIdx;
    }
    //使能软件通道的 uDMA 中断
    ROM_IntEnable(INT_UDMA);
    //禁止 μDMA 软件通道的属性设置
    ROM_uDMAChannelAttributeDisable(UDMA_CHANNEL_SW,
                                    UDMA_ATTR_USEBURST | UDMA_ATTR_ALTSELECT |
                                    (UDMA_ATTR_HIGH_PRIORITY |
                                    UDMA_ATTR_REQMASK));
    //配置 SW 通道的控制参数
    ROM_uDMAChannelControlSet(UDMA_CHANNEL_SW | UDMA_PRI_SELECT,
                              UDMA_SIZE_32 | UDMA_SRC_INC_32 | UDMA_DST_INC_32 |
                              UDMA_ARB_8);
    //设置 SW 通道的传输参数,SW 必须采用自动模式
    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_SW | UDMA_PRI_SELECT,
                               UDMA_MODE_AUTO, g_ui32SrcBuf, g_ui32DstBuf,
                               MEM_BUFFER_SIZE);
    //使能 SW 的 μDMA 通道
    ROM_uDMAChannelEnable(UDMA_CHANNEL_SW);
    ROM_uDMAChannelRequest(UDMA_CHANNEL_SW);
}
// ****************************************************************************
//配置 UART 及其引脚
// ****************************************************************************
void
ConfigureUART(void)
{
     …(省略)
}
// ****************************************************************************
//主程序,实现 DMA 传输的乒乓模式
// ****************************************************************************
int
main(void)
{
    static uint32_t ui32PrevSeconds;
    static uint32_t ui32PrevXferCount;
    static uint32_t ui32PrevUARTCount = 0;
    uint32_t ui32XfersCompleted;
    uint32_t ui32BytesTransferred;

    ROM_FPULazyStackingEnable();
    //设置系统时钟微 50 MHz
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
```

```
                    SYSCTL_XTAL_16MHZ);
//在 CPU 处于睡眠时使能外设操作
ROM_SysCtlPeripheralClockGating(true);
//使能连接板载 LED 的 GPIOF 端口
ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
//使能连接 LED(PF2)的 GPIO 引脚
ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,GPIO_PIN_2);
//初始化 UART
ConfigureUART();
UARTprintf("\033[2JuDMA Example\n");
//显示时钟频率
UARTprintf("Tiva C Series @ %u MHz\n\n",ROM_SysCtlClockGet()/1000000);
//统计显示标题
UARTprintf("CPU     Memory     UART      Remaining\n");
UARTprintf("Usage   Transfers   Transfers   Time\n");
//配置 SysTick 使之每秒发生 100 次作为时间参考。使能 SysTick 产生中断
ROM_SysTickPeriodSet(ROM_SysCtlClockGet()/SYSTICKS_PER_SECOND);
ROM_SysTickIntEnable();
ROM_SysTickEnable();
//初始化 CPU 使用率测量程序
CPUUsageInit(ROM_SysCtlClockGet(),SYSTICKS_PER_SECOND,2);
//在系统级使能 μDMA 控制器。使能处理器处于睡眠时继续运行
ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);
ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_UDMA);
//使能 μDMA 错误中断
ROM_IntEnable(INT_UDMAERR);
//使能 μDMA 控制器
ROM_uDMAEnable();
//指向通道控制结构的控制表
ROM_uDMAControlBaseSet(ui8ControlTable);
//初始化存储器到存储器的 μDMA 传输
InitSWTransfer();
//初始化 UART 的 μDMA 传输
InitUART1Transfer();
//记录当前 SysTick 的秒计数值
ui32PrevSeconds = g_ui32Seconds;
//记录当前存储器缓冲区的计数值
Remember the current count of memory buffer transfers.
ui32PrevXferCount = g_ui32MemXferCount;
//循环等待直到按钮按下。处理器在这个循环中处于休眠状态,以便 CPU 的使用率可以被测量
while(1)
{
    //检查是否耗时 1 s。如果是,则更新
    //updates.
    if(g_ui32Seconds != ui32PrevSeconds)
    {
        //点亮绿色 LED
        GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2,GPIO_PIN_2);
        //在 PuTTY 上显示 CPU 使用的百分比,其小数部分被忽略
```

```
UARTprintf("\r%3d%%     ",g_ui32CPUUsage >> 16);
//记录新的秒计数值
ui32PrevSeconds = g_ui32Seconds;
//计算从上一秒到当前传输了多少内存
ui32XfersCompleted = g_ui32MemXferCount - ui32PrevXferCount;
//记录新的传输计数值
ui32PrevXferCount = g_ui32MemXferCount;
//计算从上一秒到当前有多少个字节的数据被传输
ui32BytesTransferred = ui32XfersCompleted * MEM_BUFFER_SIZE * 4;
//显示内存传输率
if(ui32BytesTransferred >= 100000000)
{
    UARTprintf("%3d MB/s    ",ui32BytesTransferred/1000000);
}
else if(ui32BytesTransferred >= 10000000)
{
    UARTprintf("%2d.%01d MB/s   ",ui32BytesTransferred/1000000,
              (ui32BytesTransferred % 1000000)/100000);
}
else if(ui32BytesTransferred >= 1000000)
{
    UARTprintf("%1d.%02d MB/s   ",ui32BytesTransferred/1000000,
              (ui32BytesTransferred % 1000000)/10000);
}
else if(ui32BytesTransferred >= 100000)
{
    UARTprintf("%3d KB/s    ",ui32BytesTransferred/1000);
}
else if(ui32BytesTransferred >= 10000)
{
    UARTprintf("%2d.%01d KB/s   ",ui32BytesTransferred/1000,
              (ui32BytesTransferred % 1000)/100);
}
else if(ui32BytesTransferred >= 1000)
{
    UARTprintf("%1d.%02d KB/s   ",ui32BytesTransferred/1000,
              (ui32BytesTransferred % 1000)/10);
}
else if(ui32BytesTransferred >= 100)
{
    UARTprintf("%3d B/s      ",ui32BytesTransferred);
}
else if(ui32BytesTransferred >= 10)
{
    UARTprintf("%2d B/s       ",ui32BytesTransferred);
}
else
{
    UARTprintf("%1d B/s        ",ui32BytesTransferred);
```

```
        }
        //计算从上一秒发生了多少次 UART 传输
        ui32XfersCompleted = (g_ui32RxBufACount + g_ui32RxBufBCount -
                              ui32PrevUARTCount);
        //记录新的 UART 传输计数值
        ui32PrevUARTCount = g_ui32RxBufACount + g_ui32RxBufBCount;
        //计算有多少个字节通过 UART 传输
        ui32BytesTransferred = ui32XfersCompleted * UART_RXBUF_SIZE * 2;
        //显示 UART 传输速率
        if(ui32BytesTransferred >= 1000000)
        {
            UARTprintf("%1d.%02d MB/s   ",ui32BytesTransferred/1000000,
                       (ui32BytesTransferred % 1000000)/10000);
        }
        else if(ui32BytesTransferred >= 100000)
        {
            UARTprintf("%3d KB/s    ",ui32BytesTransferred/1000);
        }
        else if(ui32BytesTransferred >= 10000)
        {
            UARTprintf("%2d.%01d KB/s   ",ui32BytesTransferred/1000,
                       (ui32BytesTransferred % 1000)/100);
        }
        else if(ui32BytesTransferred >= 1000)
        {
            UARTprintf("%1d.%02d KB/s   ",ui32BytesTransferred/1000,
                       (ui32BytesTransferred % 1000)/10);
        }
        else if(ui32BytesTransferred >= 100)
        {
            UARTprintf("%3d B/s      ",ui32BytesTransferred);
        }
        else if(ui32BytesTransferred >= 10)
        {
            UARTprintf("%2d B/s       ",ui32BytesTransferred);
        }
        else
        {
            UARTprintf("%1d B/s        ",ui32BytesTransferred);
        }
        //打印纺丝线
        UARTprintf("%2ds",10 - ui32PrevSeconds);
        //熄灭绿色 LED
        GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2,0);
    }
    //使处理器进入休眠状态
    ROM_SysCtlSleep();
    //如果运行时间大于等于 10,则退出循环
    if(g_ui32Seconds >= 10)
```

```
            {
                break;
            }
        }
        //在 PuTTY 上显示停止信息
        UARTprintf("\n 停止\n");
        //无限循环且 CPU 不进入休眠状态,以便可进行调试
        while(1)
        {
            //点亮绿色 LED
            GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2,GPIO_PIN_2);
            //延时 1 位
            SysCtlDelay(SysCtlClockGet()/20/3);
            //熄灭绿色 LED
            GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2,0);
            //延时 1 位
            SysCtlDelay(SysCtlClockGet()/20/3);
        }
    }
```

2）在 C:\ti\TivaWare_C_Series－2.0.1.11577\examples\boards\EK－TM4C123GXL\目录下导入 udma_demo 工程，如图 13-5 所示。

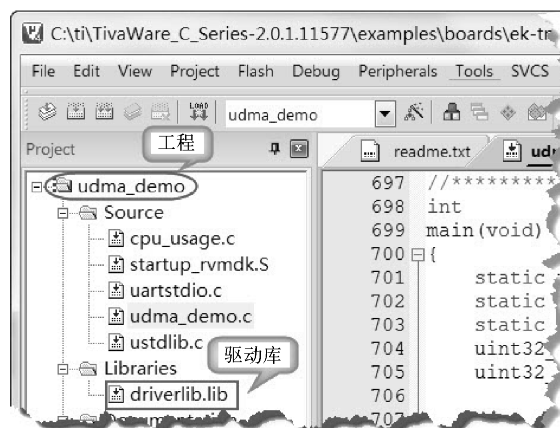


图 13-5　导入的 udma_demo 工程

3）编译 udma_demo 工程并下载到 EK－TM4C123GXL 开发板中，如图 13-6 所示。



图 13-6　将 . axf 可执行文件下载到 EK－TM4C123GXL 板中

4）程序功能测试。

① 打开 PuTTY，然后按 EK－TM4C123GXL 板上的复位键，观察到的结果如图 13－7 所示。

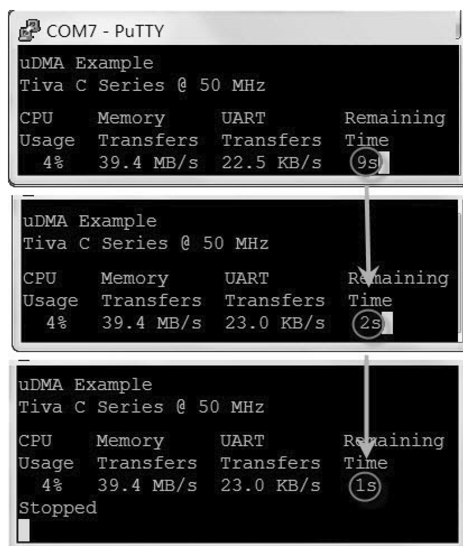


图 13-7　在 PuTTY 上观察到的程序运行结果

② 当保留时间从 9 s 递减到 1 s 过程中，可以观察到板上的绿色 LED 灯不断闪烁。

③ 从上述在板中对程序的测试结果来看，本小节所介绍的 DMA 程序实现了所需的要求，这就验证了程序的正确性。

第 14 章

通用串行总线控制器（USB）

本章简要介绍通用 USB 控制器的协议与基本架构，包括 TM4C123GH6PM USB 控制器的特点，较详细地介绍 DriverLib 库中 USB 固件函数的功能与 USBlib 固件库的结构与特点，并以 TI 的一个 USB bulk 设备例程来介绍 USB 的编程与测试方法。

本章的主要内容：

- USB 简介
- TM4C123GH6PM USB 控制器
- USB 控制器固件库函数（请见书后附录 L）
- 例程



14.1 USB 简介

1. USB 概述

USB，为 Universal Serial BUS（通用串行总线）的缩写，是一个外部总线标准，用于规范 PC 与外部设备的连接和通信。USB 接口只需要四根连线，即 2 根数据线、1 根电源线、1 根地线，数据以串行方式传输。已有 3 个版本的 USB 协议：USB1.1（速率 12 Mbit/s）、USB2.0（速率 480 Mbit/s）和 USB3.0（速率 4.8 Gbit/s），但它能达到的最大传输速率却取决于总线上最慢的“设备”，这包括主机、HUB9（集线器）和 USB 功能设备。USB 具有支持热拔插、应用广泛且使用方便、连线灵活以及独立供电等优点。USB 采用差分信号来传输数据，USB 由主机、设备和互连三部分组成。其中“主机”是指提供 USB 接口及接口管理的硬件、软件与固件的集合，它可以是 PC 或特殊的 USB 设备。在 USB 系统中仅允许一个 USB 主机存在；而“设备”包括 Hub 和功能，其最多可支持 127 个设备；“互连”是指 USB 设备与主计算机的连接和通信的方式，包括总线拓扑结构、内层关系、数据流模型和 USB 调度表等。USB2.0 使用轮询的广播方式来进行数据传输，任何数据包的发送都必须由主机来启动，在 USB 控制器中的任何时刻只允许一个数据包在执行传输任务。

注释：本小节内容包含来自网络、参考文献、USB 手册中有关 USB 的基础知识。

2. USB 常用术语

（1）管道（Pipe）

管道是主机与设备端点数据传输的连接通道，表征主机的数据缓冲区与设备端点之间交换数据的能力。管道包括数据流管道和消息管道，USB 设备上电就存在一个信息管道，使

USB 主机可通过此管道来获取设备的描述、配置、状态，并对设备进行配置。

（2）端点（Endpoint）

端点是 USB 设备中可进行数据收发的最小单元，支持单向或者双向的数据传输。USB 控制器可提供两个专用的控制端点（即输入："IN" 和输出："OUT"）以及可用于与主机进行通信的 30 个可配置的端点（即 15 个 IN 端点和 15 个 OUT 端点）。

（3）包格式（Packet）

它是 USB 总线上数据传输的基本形式，在 USB 总线上传输的数据都是先打包后进行传输的。它包括 SOP（包起始）、SYNC（包同步）、包内容和 EOP（包结束）4 个部分。其中"包内容"由 PID 字段、地址字段、帧号字段、数据字段和 CRC 字段等中的一个或多个字段组成，如图 14-1 所示。



图 14-1　包格式

（4）包的分类

① 令牌包：它决定数据包的传输类型，其包含 PID 字段、地址字段和 CRC 字段。在 USB 控制器中，仅主机可发出令牌包，它是事务处理的第一阶段。较为重要的令牌包是 IN/OUT/SETU。

② 帧首包：它包含 PID 字段、帧号字段和 CRC 字段。对于低速/全速通信每隔 1 ms 发送一个 SOF，而对于高速通信每隔 125 μs 发出一个帧首包。

③ 数据包：它包含 PID 字段、数据字段和 CRC 字段，并以 8 字节数据字段传输。PID 指定是 DATA0 还是 DATA1 用于数据的重发管理；数据字段为 0～1023 个字节；CRC 字段为 16 字节，用于对数据字段进行校验。

④ 握手包：握手包确认传输是否成功，其仅包含 PID 字段，是最简单的信息包。

（5）事务（Transaction）

事务是指把一个"服务"传送到一个"端点"的处理过程，其中"服务"是指主机发送到设备的数据，或主机从设备接收的数据。事务包括：

① IN（输入）事务：即总线上的 USB 设备向 USB 主机发送数据包的处理过程。

② OUT（输出）事务：即 USB 主机向总线上的 USB 设备发送数据包的处理过程。

③ SETUP 事务：仅在 USB 控制传输中用于对 USB 设备进行配置。

（6）单向传输

单向传输是指在某个端点上只支持一个方向的传输，即要么是输出，要么是输入。若要在两个方向上进行单向传输，则需要占用两个端点并分别进行配置。

（7）集线器

它提供把标准的多个 USB 设备连接到主机控制器的功能。

（8）设备

设备是指 USB 设备，由一个或多个配置组成。可用设备描述符来表达设备的总体信息。

（9）配置

一个 USB 设备可用一个或多个配置来说明。在使用 USB 设备之前，需为该设备指定一个或多个合适的配置。且可通过配置描述符来说明。

（10）接口

接口为断点的集合，可包含在一个或多个配置中。可通过接口描述符来说明设备中的接口特性。

（11）USB OTG

USB OTG 即 USB On－The－Go 的缩写，它用于在没有主机的情况下，实现设备间的数据传输。

3. USB 拓扑结构

USB 的拓扑结构如图 14-2 所示。

从图 14－2 可看到，分层星形拓扑结构，可使网络中的每个集线器都处于星型的中心，每条线段都是点对点的连接，以实现 USB 的物理连接，电缆的最大长度为 5 m。当以主机－根集线器（HOST－ROOT HUB）作为起点时，USB 最多可支持 7 个分层（图 14-2 中仅画出了 6 个层），即 USB 系统中最多可包含 5 个 HUB 的级联。一个复合设备（Compound Device）将同时占有两个或更多的层。

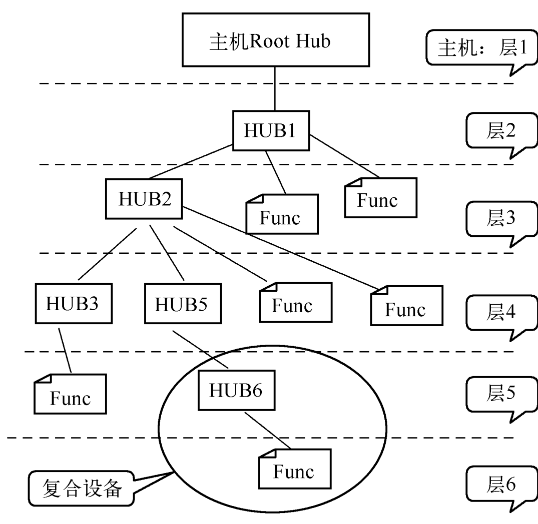


图 14-2　USB 的分层星形拓扑结构

4. USB 总线信号

为了抗干扰和使信号同步，在 USB 总线上传输的信号，一般采用如图 14-3 所示的 NRZI（反向不归零码）编码差分信号，并且需在 NRZI 编码前，在数据中的恰当位置添 0 来保持信号的同步。

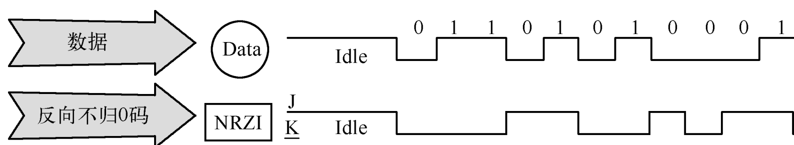


图 14-3　NRZI 编码差分信号

USB 电缆信号如图 14-4 所示。

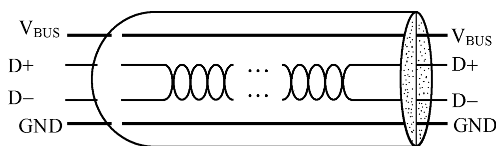


图 14-4　USB 电缆信号

其中，D＋和 D－信号即为 NRZI 编码差分信号、VBUS 为＋5 V 电源线、GND 为地线。

5. USB 通信管道

在 USB 控制器的分层结构中，数据的传输是通过管道进行通信的，即通过主机的系统/客户软件在 USB 设备的各端点之间进行的，如图 14-5 所示。

6. USB 实现

USB 的实现如图 14-6 所示。

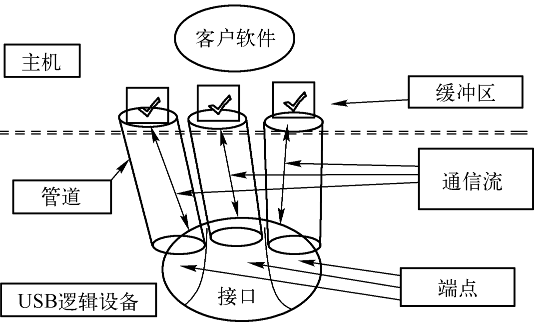


图 14-5　USB 的通信管道

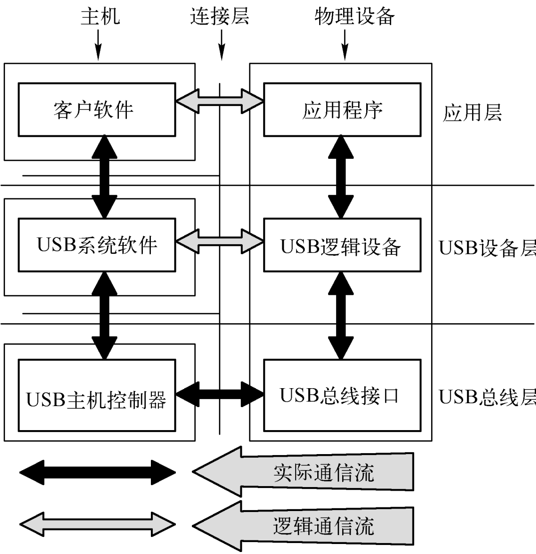


图 14-6　USB 的实现

在图 14-6 中可以看到，USB 控制器却可分为 3 个层次，即应用层、USB 设备层和 USB 总线层。每层都由不同的功能模块构成来简化通信的复杂度，其中 USB 总线层用来执行主机和设备之间数据传输的物理层实现和总线管理；USB 设备层对于 USB 系统软件是透明的，依据其所见的设备层来执行对设备的操作；应用层可以利用相应的客户软件向主机提供一些附加的功能。而 USB 设备层和应用层的通信只是逻辑上的，真实的物理通信由 USB 总线层来执行。各单元的功能介绍如下：

（1）客户软件

客户软件运行于 USB 设备的主机端，来自于操作系统或 USB 设备提供商。

（2）USB 系统软件

USB 系统软件为 USB 设备提供支持的特定操作系统。该软件通常由操作系统提供，独立于特定 USB 设备或客户端软件。

（3）USB 主机控制器

把 USB 设备要连接到一个主机的硬件和软件。

（4）物理设备

在 USB 设备端执行某些有用的客户程序。

7. 常用的 USB 类

1）人机接口类（HID）。

① 不需要开发 PC 驱动就可完成 USB 设备的开发（驱动由 Windows 操作系统支持）。

② 可提供多种预定义的通信类型。
③ 具有高度的灵活性。
2）通信类（CDC）。
3）音频类（Audio）。
4）设备固件升级类（DFU）。
① 允许设备通过与 USB 主机的连接来完成其固件的升级。
② 允许设备在其设备描述符中通告其升级能力。
③ 在组合设备中，DFU 主要作为设备的子类，可为其他设备类增添升级功能。
④ 提供一种更加通用的 USB 启动加载程序。
5）大容量存储设备类（MSC）。
① 允许微控制器对 USB 大容量存储媒设备（如闪存）进行文件的读/写。
② 文件系统由开源程序 FatFs 提供。

注意： *微软已经为这 5 类提供了驱动程序，TI 也为其提供了演示例程。*

8. 传输中的包含关系

1）传输（Transfer）⊇N 个事务（Transaction），N＝1，2，3…。
即 USB 总线上的一个“传输”由一个或多个“事务”组成。
2）事务（Transaction）⊇N 个包（Packet），N＝1，2，3…。
即一个“事务”由一个或多个“包”组成。

9. 传输类型

（1）控制传输

用于可靠的、突发与非周期性，由主机软件发出的请求或回应用于命令和状态的传输。

（2）中断传输

用于小规模数据、周期性与低速，允许固定延迟的通信。

（3）同步传输

对于周期与持续性的通信，用于传输与时效相关的信息，并且在数据中保存时间戳信息。

（4）批量传输

用于非周期性、批量可靠数据的通信，数据可占用任意带宽，当这些数据没有可用带宽时，可容忍等待。

10. 描述符（Descriptor）

描述符是一种完整的数据结构，可以通过 C 语言来编程实现。用于描述 USB 设备的功能、特性、类型和对资源的需求等信息。其目的是通过各种描述符以问答的方式让主机了解每个设备的功能。只有主机确认了这些信息之后，才能使设备正常工作。常用的描述符如下：

（1）设备描述符

它是设备在连接到主机时读取的第一个描述符，它定义了 USB 设备类、制造商 ID、产品 ID 和若干个配置等总体信息，一个 USB 设备仅包含一个设备描述符，并且该描述符决定了设备可获得的“配置”个数，见表 14-1。

表 14-1　设备描述符

偏移量	字　段	大小	值	描　述
0	bLength	1	数字	描述符的字节数
1	bDecriptorType	1	常量	设备类型限定符（0x01 = 设备描述符）
2	bcdUSB	2	BCD	USB 规范发布版本号
4	bDeviceClass	1	类	类代码
5	bDeviceSubClass	1	子类	子类代码
6	bDevicePortocol	1	协议	协议代码
7	bMaxPacketSize0	1	数字	端点 0 的最大包大小（仅 8、16、32、64 为合法值）
8	idVendor	2	ID	厂商标志（由 USB - IF 组织赋值）
10	idProduct	2	ID	产品标志（由厂商赋值）
12	bcdDevice	2	BCD	设备发布版本号
14	iManufacturer	1	索引	厂商信息的字符串描述符的索引值
15	iProduct	1	索引	产品信息的字串描述符的索引值
16	iSerialNumber	1	索引	设备序列号信息的字串描述符的索引值
17	bNumConfigurations	1	数字	可能的配置数目

（2）配置描述符

用于定义设备的配置信息，如接口总数、最大电源消耗等。一个 USB 设备至少可包含一种配置，它决定了设备的特性和能力。并且通过描述符可获取设备“接口”的个数，见表 14-2。

表 14-2　配置描述符

偏移量	字　段	大小	值	描　述
0	bLength	1	数字	以字节为单位的描述符长度（0x09）
1	bDescriptorType	1	常量	配置描述表类型（0x02）
2	wTotalLength	2	数字	返回的配置数据的总长。包括所有描述符的组合长度（即配置、接口、端点和设备类及厂商定义）
4	bNumInterfaces	1	数字	配置所支持的接口数目
5	bCongfigurationV alue	1	数字	在 SetConfiguration( ) 请求中用作参数来选择该配置的值
6	iConfiguration	1	索引	描述该配置的字串描述表索引
7	bmAttributes	1	位图	自供电/远程唤醒及总线供电配置
8	MaxPower	1	mA	以 2 mA 为单位的总线功耗

（3）接口描述符

描述一个接口所提供的配置，如接口类型、使用了何种非 0 端点等。该描述符包含类、子类与协议的信息，并且通过它可获取设备“端点”的个数，见表 14-3。

表 14-3　接口描述符

偏移量	字　段	大小	值	说　明
0	bLength	1	数字	以字节为单位的描述符长度（0x09）
1	bDescriptorType	1	常量	接口描述符类型（0x04）
2	bInterfaceNumber	1	数字	用于标识接口的编号
3	bAlternateSetting	1	数字	用来选择优先字段中标识接口切换设置的值
4	bNumEndpoints	1	数字	接口所用的端点个数（端点 0 除外）

（续）

偏移量	字　段	大小	值	说　明
5	bInterfaceClass	1	类	类代码
6	bInterfaceSubClass	1	子类	子类代码
7	bInterfaceProtocol	1	协议	协议代码
8	iInterface	1	索引	接口字串描述符的索引

（4）端点描述符

接口的每个端点都有自己的描述（端点 0 除外），它用于定义每个端点的传输类型与速率等，见表 14-4。

表 14-4　端点描述符

偏移量	字　段	大小	值	说　明
0	bLength	1	数字	以字节为单位的描述符长度（0x07）
1	bDescriptorType	1	常量	端点描述符类型（0x05）
2	bEndpointAddress	1	端点	描述的 USB 设备上端点的地址与方向
3	bmAttributes	1	位图	选择传输类型与附加信息
4	wMaxPacketSize	2	数字	端点所能接收或发送的最大数据包的大小
6	bInterval	1	数字	数据传输时端点轮询的间隔或 NAK 率

（5）字符描述符（可选）

用于提供一些便于阅读的信息，见表 14-5。

表 14-5　字符描述符

偏移量	字　段	大小	值	描　述
0	bLength	1	N+2	以字节为单位的描述符长度（可变）
1	bDescriptorType	1	常量	字串描述符类型（0x03）
2	wLANGID[0]	2	数字	语言标识符（LANGID）代码 0
…	…	…	…	…
N	wLANGID[x]	2	数字	语言标识符（LANGID）代码 x

11. USB 设备的状态转移图

USB 设备的状态转移图如图 14-7 所示。

12. USB 设备的枚举

枚举是 USB 控制器中的重要活动，它由 8 字节的标准请求组成。通过枚举主机可以获得设备的各种描述信息，如 PID、VID、设备分类、配置数量、端点个数等。主机根据这些信息加载合适的设备驱动程序，并对设备进行合理的配置，以实现 USB 设备的特定功能（可对比图 14-7 的 USB 状态转移图）。USB 设备的枚举过程归纳如下：

① 供电→② 复位→③ 获取设备描述符前八个字节数据→④ 复位（可选）→⑤ 分配地址→⑥ 获取设备描述符（重新获取设备描述符来得到设备的总体信息）→⑦ 获取配置描述符→⑧ 获取字符串描述符（可选）→⑨ 配置。

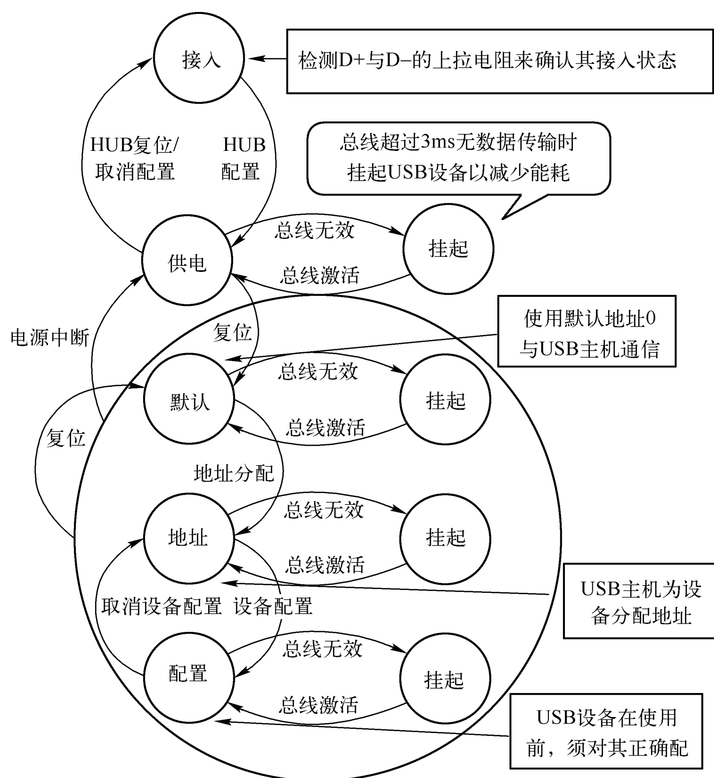


图 14-7　USB 状态转移图



14.2　TM4C123GH6PM USB 控制器

TM4C123GH6PM USB 控制器支持 USB 主机/设备/OTG 功能，在点对点通信过程中可作为全速和低速的功能控制器。满足 USB2.0 标准，包含挂起和恢复信号。16 个端点包含 2 个用于控制传输的专用连接端点（即一个 IN 和一个 OUT）与 14 个由固件定义的端点，并带有一个大小可动态变化的 FIFO 来支持多包队列。采用 μDMA 访问 FIFO 的方法，将对系统软件的依赖度降至最小。USB 设备启动方式灵活，可软件控制是否在启动时连接，并遵守 OTG 标准的会话请求协议（SRP）和主机协商协议（HNP）。

该驱动程序包含在 DriverLib/usb.c 中，DriverLib/usb.h 包含了 USB 控制器 API 函数的定义。

14.2.1　USB 的特点

TM4C123GH6PM USB 模块有如下特性：

1）符合 USB-IF 认证标准。

2）支持 USB 2.0 全速（12 Mbit/s）和低速模式（1.5 Mbit/s）操作。

3）集成 PHY。

4）连接电源管理支持使用链路状态感知，以减少能耗。

5）4 种传输类型：控制传输、中断传输、批量传输和同步传输。

6）16 端点。

① 1 个专用的 IN 控制端点和 1 个专用 OUT 控制端点。

② 7 个可配置的 IN 端点和 7 个可配置的 OUT 端点。

7）4 KB 专用端点存储空间，1 个端点可定义为双缓存 1023 字节同步传输。

8） 支持 VBUS 电压下降（droop） 和有效 ID 检测，以及发出中断。

9） 采用 μDMA 提高传输数据有效。

① 用于发送和接收的独立通道多达 3 个 IN 端点和 3 个 OUT 端点。

② 当发送 FIFO 中包含所需数量的数据后，可产生通道请求。

14.2.2 USB 模块框图

TM4C123GH6PM USB 的模块框图如图 14-8 所示。

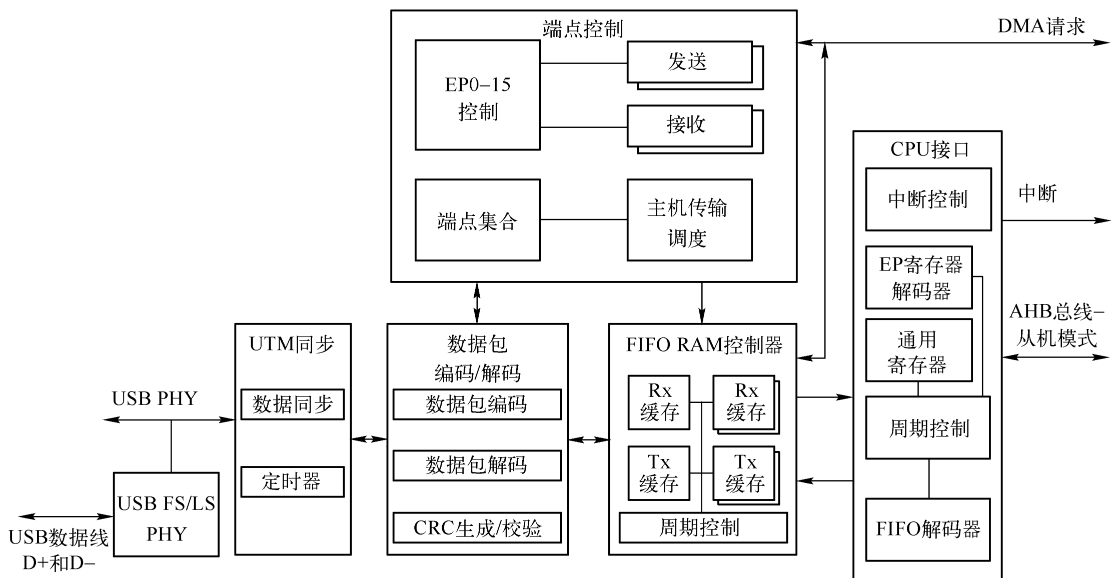


图 14-8　USB 模块框图

14.2.3 USB 信号描述

USB 信号描述见表 14-6。

表 14-6　USB 信号（64LQFP）

引脚名称	引脚编号	引脚复用/赋值	引脚类型	缓冲区类型	描　　述
USB0DM	43	PD4	I/O	模拟	USB0 的双向差分数据引脚（USB 中的 D－信号）
USB0DP	44	PD5	I/O	模拟	USB0 的双向差分数据引脚（USB 中的 D＋信号）
USB0EPEN	5 14 63	PF4（8） PC6（8） PD2（8）	O	TTL	在主机模式中，可选的向 USB 总线供电的外部电源控制

（续）

引脚名称	引脚编号	引脚复用/赋值	引脚类型	缓冲区类型	描　　述
USB0ID	45	PB0	I	模拟	该信号用于检测 USB ID 信号的状态。此时 USB PHY 将在使能一个集成上拉电阻，通过外部元件（USB 连接器）检测 USB 控制器的初始状态（即电缆的 A 侧配置为下拉电阻，B 侧配置为上拉电阻）
USB0PFLT	13 64	PC7（8） PD3（8）	I	TTL	在主机模式中，可选的由外部电源管理芯片来接收电源的错误状态指示
USB0VBUS	46	PB1	I/O	模拟	此信号用于会话请求协议。USB PHY 可通过此信号检测 VBUS 的电平，并在 VBUS 脉冲期间瞬间上拉

14.2.4 USB 功能描述

TM4C123GH6PM USB 控制器支持 OTG 标准的会话请求协议和主机协商协议，提供完整的 OTG 协商会话请求协议，允许连接在 USB 电缆 B 端的 B 类设备向连接在 A 端的 A 类设备发出请求，通知 A 类设备打开 VBUS 电源。主机协商协议用于在启动会话请求协议上电后，决定 USB 电缆哪端的设备作为 USB 主机。当该设备连接到非 OTG 外设或设备时，控制器可以检测出电缆终端所使用的设备类型，并提供一个寄存器来指示该控制器是用作主机，还是用作设备控制器。以上的操作都是由 USB 控制器自动处理的。基于这种自动探测机制，系统使用单个的 A/B 型连接器取代 A 型或 B 型连接器，可支持与另外的 OTG 设备完整的 OTG 协商。

另外，USB 控制器还提供对连接到非 OTG 的外设或主机控制器的支持。可将 USB 控制器设置为专用的主机或设备模式，此时，USB0VBUS 和 USB0ID 引脚可被设置成 GPIO 功能使用。但当 USB 控制器用作自供电设备时，必须将 GPIO 输入引脚或模拟比较器输入引脚连接到 VBUS 脚上，并配置为在 VBUS 掉电时将发出中断。该中断用于禁止 USB0DP 信号的上拉电阻。

注意：当使用 USB 模块时，时钟源 MOSC 无论使用 PLL，系统时钟应至少为 20 MHz。

1. 作为设备时的操作

本节将描述 TM4C123GH6PM USB 控制器用作 USB 设备时的行为，包括：IN 端点、OUT 端点、进入和退出挂起模式、帧起始。其中，IN 事务通过使用端点的发送端点寄存器，由端点的发送接口进行控制。OUT 事务通过使用端点的接收端点寄存器，由端点的接收接口进行控制。

当配置端点的 FIFO 大小时，需要考虑最大数据包的大小。

① 批量传输：批量端点的 FIFO 可配置为最大包长（64 个字节），如果使用双包缓存，则需要配置为 2 倍于最大包长大小。

② 中断传输：中断端点的 FIFO 可配置为最大包长，如果使用双包缓存，则需要配置为两倍的最大包长大小。

③ 同步传输：同步端点的 FIFO 比较灵活，最大可支持 1023 字节。

④ 控制传输：也可以用于为 USB 设备指定一个独立的控制端点。但是在大多数情况下，USB 设备应该在 USB 控制器的端点 0 上使用专用的控制端点。

（1）端点

USB 控制器提供两个专用的控制端点（即 IN 和 OUT）与可用于跟主机控制器进行通信的 14 个可配置的端点（即 7 个 IN 和 7 个 OUT）。端点号和方向与对应的相关寄存器直接关联。例如，在主机向端点 1 发送数据时，所有的配置和数据都存在端点 1 的发送寄存器接口中。

端点 0 是专用的控制端点，用于在枚举期间所有对端点 0 的控制传输事务，或对端点 0 的其他控制请求。端点 0 使用 USB 控制器 FIFO RAM 中的首 64 字节，该内存对于 IN 传输事务和 OUT 传输事务是共享的。剩余的 14 个端点可配置为控制、批量、中断或同步端点。其中 7 个配置为 IN 端点，另外 7 个配置为 OUT 端点。这些成对的端点类型可以不同。例如，成对端点的 OUT 部分可配置成批量端点，IN 部分可配置成中断端点。连接到每个端点的 FIFO 的地址和大小可根据实际需求来修改。

（2）IN 事务

IN 数据传输可通过发送端点的 FIFO 来处理。7 个可配置 IN 端点的 FIFO 大小，由 USB 发送 FIFO 的起始地址寄存器（USBTXFIFOADD）决定。传输时发送端点 FIFO 中的最大数据包长可编程配置，其大小由写入该端点的 USB 端点 n 的发送最大发送数据寄存器（USBTXMAXPn）中的值决定。端点的 FIFO 既可配置成双包缓存，也可配置成单包缓存。当双包缓存使能时，在 FIFO 中可缓冲两个数据包，要求 FIFO 的容量应不小于两个数据包的大小。当禁止双包缓存时，即使数据包长小于 FIFO 容量的一半，也仅能缓冲一个数据包。

（3）OUT 事务

OUT 事务的数据可通过接收端点的 FIFO 来处理。7 个可配置 OUT 端点的 FIFO 大小，由 USB 输出 FIFO 的起始地址寄存器（USBRXFIFOADD）决定。任何接收端点 FIFO 中的最大数据包长都可编程配置，其大小由写入该端点的 USB 端点 n 的最大接收数据寄存器（USBRXMAXPn）中的值决定。当双包缓存使能时，在 FIFO 中可缓冲两个数据包。当禁止双包缓存时，即使数据包长小于 FIFO 容量的一半，也仅能缓冲一个数据包。

（4）调度

设备无权控制事务调度，而只能由主机来调度决定。TM4C123GH6PM USB 控制器可随时建立传输事务。USB 等待从主机发出的请求以及当事务传输完成或由于某些错误而终止时，都会产生一个中断。如果主机发起请求，而设备还未准备就绪，那么 USB 将对所有请求发送一个忙响应（NAK）信号，直到其准备就绪。

（5）挂起

当 USB 总线超过 3 ms 时间无数据传输时，它将自动进入挂起（SUSPEND）模式。如果在 USB 中断使能寄存器（USBIE）中已使能挂起中断，这时将会产生一个中断。当 USB 控制器处于挂起模式下，USB PHY 也可进入挂起模式。当检测到恢复信号时，USB 将退出挂起模式，并且 USB PHY 也将退出挂起模式。如果已使能了恢复中断，则将产生一个中断。可通过配置 USB 电源寄存器（USBPOWER）中的 RESUME 位来强制 USB 控制器退出挂起模式。当该位被置位，使 USB 控制器退出挂起模式，同时恢复信号将出现在总线上。RESUME 位必须在 10 ms（最大 15 ms）后被清零以结束恢复信号。

注意：为了满足 USB 电源要求，控制器可以进入深度休眠模式，使控制器保持在静止状态。休眠模式不应该用于挂起模式，因为所有的内部状态信息在处于休眠状态时将会丢失。

（6）帧起始

在设备模式时，其每 1 ms 将接收到由主机发出的帧起始包（SOF）1 次。在收到 SOF 数据包时，数据包中的 11 位帧号将被写入到 USB 帧值寄存器（USBFRAME）中，且发出 SOF 中断信号（由应用程序来处理）。一旦 USB 控制器开始接收 SOF 包，预计每 1 ms 接收 1 次。如果 1.00358 ms 后仍未收到 SOF 包，可认为该包已丢失，并且 USBFRAME 寄存器也不会被更新。在重新成功接收这些数据包时，USB 控制器将继续并重新同步这些脉冲来接收 SOF 包。

（7）USB 复位

在检测到 USB 总线上的复位条件时，USB 控制器将会自动执行以下操作：

① 清除 USBFADDR 寄存器。

② 清除 USB 端点索引寄存器（USBEPIDX）。

③ 刷新所有端点 FIFO。

④ 清除所有的控制/状态寄存器。

⑤ 使能所有端点中断。

⑥ 产生一个复位中断。

在应用软件驱动的 USB 控制器中收到复位中断时，将关闭任何打开的管道，并等待 USB 控制器开始总线枚举。

（8）连接/断开

连接到 USB 总线上的 USB 控制器将由软件来处理。USB PHY 通过置位和清零 USB-POWER 寄存器中的 SOFTCONN 位，使其可在正常模式和无驱动模式之间切换。当 SOFT-CONN 位置位时，USB PHY 为正常模式，USB 总线上的 USB0DP/USB0DM 线被使能。同时，USB 控制器不会响应任何信号（USB 复位除外）。

在 SOFTCONN 位清零时，USB PHY 为无驱动模式，USB0DP 和 USB0DM 为高阻态，这时，对于 USB 总线上的其他设备，USB 控制器为断开状态。因无驱动模式为默认状态，所以 USB 控制器只有在 SOFTCONN 位置位时才会被接通。然后，应用软件可以选择何时设置 PHY 进入正常模式。系统将经历一个较长时间来初始化程序，以确保初始化完成。在连接 USB 总线之前，为系统的设备枚举做好准备。一旦 SOFTCONN 位置位，USB 控制器可通过清除该位来断开连接。

注意：当设备连接到主机时，USB 控制器不会产生中断。不过当主机终止会话时，将会产生一个中断。

2. 作为主机时的操作

当 TM4C123GH6PM 控制器运行在主机模式时，在 USB 由设备变为主机或由主机变为设备前，必须通过设置软件复位控制寄存器 2（SRCR2）中的 USB0 位来复位 USB 控制器。USB 支持全速和低速设备，支持点对点通信和通过集线器操作。允许 USB 2.0 集线器使用低速设备和全速设备。支持控制传输、批量传输、同步传输和中断传输。本小节描述当 USB 在主机模式时的操作，但与设备操作类似的部分，这里不再累述。

（1）端点

专用的控制接口只用于与设备的端点 0 之间的控制传输。其用于设备枚举或其他使用该端点的控制功能。控制端点的 IN 和 OUT 事务共享 USB FIFO 内存中的首 64 字节。其余 IN

和 OUT 接口可配置为与控制端点、批量端点、中断端点或同步端点通信。

这些 USB 接口可同时调度，可用于与任何设备的任何端点的 7 个独立的 OUT 事务和 7 个独立的 IN 事务。并用三组成对的寄存器来控制 IN 和 OUT 端点。通过配置可与不同类型的端点以及不同设备的不同端点进行通信。例如，第一对端点可分开控制，即输出部分与设备的批量输出端点 1 通信，输入部分与设备的中断端点 2 通信。

无论用于点对点通信还是通过集线器通信，在访问设备前，都必须设置相关的 USB 端点 n 的接收功能地址寄存器（USBRXFUNCADDRn）或 USB 端点 n 发送功能地址寄存器（USBTXFUNCADDRn），以记录每个接收和发送端点将要访问的设备地址。

USB 控制器支持通过 USB 集线器连接设备，这可通过一个寄存器实现，该寄存器说明集线器地址和每个传输的端口。FIFO 的地址和大小可定制，并可指定用于任一 USB 输入和输出的传输。

（2）IN 事务

IN 事务的处理采用与设备模式处理 OUT 事务类似的方式，但传输事务必须通过设置寄存器 USBCSRL0 中的 REQPKT 位开始，向事务调度指示该端点存在一个激活的传输。此时事务调度向目标设备发送 1 个输入令牌包。当接收到数据包且存放到接收 FIFO 中时，USBCSRL0 寄存器中的 RXRDY 位置位，同时产生相应的接收端点中断信号，指示有 1 个数据包需要从 FIFO 中读出。

在数据包读出时，必须将 RXRDY 位清零。USBRXCSRHn 寄存器中的 AUTOCL 位可用于在最大包从 FIFO 中读出时将自动清零 RXRDY 位。在 RXRDY 位清零时，USBRXCSRHn 寄存器中的 AUTORQ 位将会使 REQPKT 位自动置位。AUTOC 和 AUTORQ 位可与 μDMA 配合使用，可实现无需处理器干预的批量传输。在 RXRDY 位清零时，控制器会向设备发出 1 条确认信息。在确定传输的数据包个数时，将在与该端点关联的 USB 端点 n 块传输请求包数量寄存器（USBRQPKTCOUNTn）中配置要传输的数据包个数。每次请求前，USB 控制器都将 USBRQPKTCOUNTn 寄存器中的值减 1。当 USBRQPKTCOUNTn 寄存器中的值减到 0 时，将清零 AUTORQ 位以阻止后面试图进行的事务。如果传输的数量未知，则应当将 USBRQPKTCOUNTn 位清零。AUTORQ 保持置位状态，直到接收到一个短包才清零，此种情形只在批量传输的末尾才可出现。

如果设备用 NAK 响应批量或中断传输的输入令牌，USB 主机将重试直到达到设置的 NAK 限制次数。如果目标设备用 STALL 握手响应，USB 主机将不重试传输，而将寄存器 USBCSRL0 中的 STALLED 位置位。如果目标设备在需要的时间内不响应输入令牌包，或包存在 CRC 或位填充错误，USB 主机将重试传输。如果三次重试，目标设备仍无响应，USB 控制器将把 REQPKT 位清零，将 USBCSRL0 寄存器中的 ERROR 位置位。

（3）OUT 事务

OUT 事务的处理采用与设备处理 IN 事务类似的方式。当包装载到发送 FIFO 中时，USBTXCSRLn 寄存器中的 TXRDY 位必须置位。若将 USBTXCSRHn 寄存器中的 AUTOSET 位置位，当最大包长的包装载到 FIFO 中时，TXRDY 位将自动置位。此外，AUTOSET 位与 μDMA 控制器配合使用，可在不需要软件干预的情况下完成批量传输。

若目标设备用 NAK 响应输出令牌包，USB 主机将重试直到达到设置的 NAK 限制次数。如果目标设备用 STALL 握手响应，USB 主机将不会重试传输，而通过将 USBTXCSRLn 寄存

器中的 STALLED 位置位来中断主处理器。若目标设备在要求的时间内不响应输出令牌包，或包存在 CRC 或位填充错误，USB 主机将重试传输。如果三次重试，目标设备仍无响应，USB 将清空 FIFO，以及把寄存器 USBTXCSRLn 中的 ERROR 位置位。

（4）事务调度

事务调度由 USB 主机自动处理。USB 主机允许根据端点事务类型配置端点通信调度。中断传输既可每 1 帧进行 1 次，也可每 255 帧进行 1 次，可在 1 ~ 255 帧之间以 1 帧的递增调度。批量端点不允许调度参数，但在设备端点不响应时，允许 NAK 超时。同步端点可在 1 ~ 216 帧之间调度（2 的幂）。

USB 维持帧计数。如果目标设备为全速设备，控制器在每帧开始时将自动发送 SOF 包，同时帧计数值加 1。若目标设备为低速设备，将在总线上发送 K 状态来保持总线激活，防止低速设备进入挂起模式。

在发送 SOF 包后，USB 主机控制器将巡检所有配置就绪的端点，寻找激活的传输事务。对于 REQPKT 位置位的接收端点或 TXRDY 或 FIFONE 位置位的发送端点，可视为存在激活的传输事务。

若传输创建在 1 帧的第一个调度周期，而且端点的间隔计数器减到 0，那么同步传输和中断传输开始。因此每个端点的中断传输和等同步传输每 n 帧才会发生 1 次，其中 n 为在 USBTXINTERVALn 或 USBRXINTERVALn 寄存器设置的间隔。

若在帧中的下一个 SOF 包前有足够时间来完成传输，则激活的批量传输会立即开始。如果传输需要重发时，需要在调度器先行检查完其他端点是否有激活的传输之后，才能重新传输。这保证了发送大量 NAK 响应的端点不会阻塞总线上的其他传输的正常进行，并且允许用户设置目标设备发送 NAK 端点的超时限制。

（5）USB 集线器

以下介绍仅可使用 USB 2.0 集线器的主机。当低速设备或全速设备通过 USB 2.0 集线器连接到 USB 主机时，集线器的地址和端口的详细信息必须记录在相应的 USB 端点 n 接收集线器地址（USBRXHUBADDRn）和接收集线器端口（USBRXHUBPORTn）或 USB 发送集线器地址（USBTXHUBADDRn）和发送集线器端口（USBTXHUBPORTn）寄存器中。此外，设备的运行速度也必须记录在 USB 类型端点 0 寄存器（USBTYPE0）、USB 端点 n 主机配置发送类型寄存器（USBTXTYPEn）或主机配置接收类型寄存器（USBRXTYPEn）中。

对于集线器通信，为了支持更多数量的设备，USB 主机允许通过简单的更新记录在这些寄存器中的地址和速度信息来动态改变这些配置。设备功能端点配置的改变必须在相关端点正在进行的传输完成后进行。

（6）OTG 模式

为了省电，USB OTG 允许当需要时才给 VBUS 上电，在不使用 USB 总线时，断开 VBUS，而 VBUS 总是由总线上的 A 设备提供。OTG 控制器通过 PHY 采样 ID 输入来决定哪个是 A 设备哪个是 B 设备。在 ID 信号拉低时，表示 OTG 控制器作为 A 设备使用；在 ID 信号为高时，表示 OTG 控制器作为 B 设备角色。OTG 的操作过程如下：

① 开始会话。

② 活动性探测。

③ 主机协商。

3. DMA 操作

USB 外设提供了连接到 μDMA 控制器的接口。μDMA 控制器提供了 3 个发送端点和 3 个接收端点的独立通道。通过 USB DMA 选择寄存器（USBDMASEL），软件可选择哪个端点要使用 μDMA 通道服务。发送和接收通道分别通过 USBTXCSRHn 和 USBRXCSRHn 寄存器来使能 USB 的 μDMA 操作。当 μDMA 操作使能时，USB 在 FIFO 可传输数据时，将在使能的接收或发送通道上发出 μDMA 请求。当任一 FIFO 可传输数据时，则该通道发出突发请求。μDMA 通道必须配置为基本模式，μDMA 传输的大小需限制为 USB FIFO 的整数倍。使用 μDMA 的 USB FIFO 的读/写传输都须这样配置。

若 USBTXCSRHn/USBRXCSRHn 寄存器中的 DMAMOD 位清零，则在 μDMA 传输中的每个包传输完成后都会产生中断信号，但 μDMA 会继续传输数据。如果 DMAMOD 位置位，整个 μDMA 传输完成后才会产生中断信号。此中断将加载到 USB 的中断向量中。因此，如果 USB 操作使用了中断和使能了 μDMA，则必须设计 USB 中断服务函数来处理该中断。



14.3 USB 固件库函数

本小节将简要介绍 USB 库的函数功能（DriverLib 库部分）及 USBlib 分层框架结构。USB 固件库包括两个部分：DriverLib 库和 USBlib 库。

14.3.1 USB 的分层框架结构

固件库分层结构可使开发人员依照自己的需求来调用不同层的 API，如图 14-9 所示。

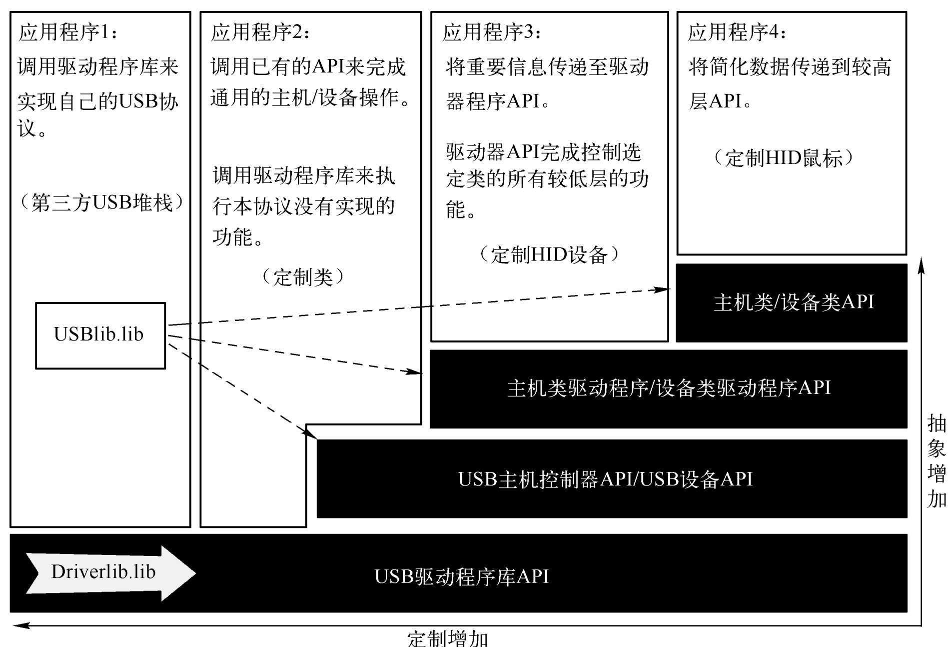


图 14-9　USB 库的框架结构

在图 14-9 中所示的 4 种可能的应用程序，将使用不同的编程接口来实现自己的 USB

功能。下面将简要介绍图中 4 层 API 函数的特点：

1）USB 驱动程序库 API（USB DriverLib API）：是 USB 设备堆栈中最底层的 USB 驱动程序，位于 DriverLib 库中的 usb.c 源代码和头文件 usb.h. 中。在图 14-9 中的“应用程序 1”提供了直接编写设备功能的 API。其实，这个 API（固件库）是上面 USB 控制器硬件寄存器中非常薄的一层，因此不提供任何更高级别的 USB 事务支持（如端点零事务处理，标准描述符和请求处理等），应用程序通常不会仅使用这个 API 来访问 USB 功能。所以，该驱动程序库在开发 USB 应用时适合作为一个接口来使用，例如，第三方的 USB 堆栈。

2）USB 设备 API（USB Device API）：USB 设备 API 在 USB 库用尽可能多的类独立代码为开发全功能的 USB 设备应用程序提供了一组专门的函数。该 API 通过从主机标准请求支持设备枚举并处理代表应用程序的端点零状态机。使用该接口的应用程序将提供在初始化期间要发布到主机的描述符，以及提供配置硬件所需的 USB 设备 API 信息。不但与异步事件有关的 USB 设备通过回调函数集来通知应用程序，而且还提供了对 USB 设备初始化的 API。该 API 用于在 USB 设备类驱动程序的开发以及提供现有类驱动程序不支持的 USB 功能应用程序的直接使用，也可以被现有的类驱动程序 API（USB Device Class Driver API）不支持的 USB 功能应用直接使用。该设备的例子需要复杂的接口切换设置。USB 设备 API 可看作是一组扩展 USB DriverLib 库的高层次 API，而不仅仅是对其进行简单的封装。因此，在开发 USB 设备 API 时，调用一些底层 USB 驱动 API 仍然是必要的。

3）设备类驱动程序 API（USB Device Class Driver API）：设备类驱动程序库提供了所需高层次特定的 USB 功能，而无需大多数的 USB 事务处理和所需的连接管理。这些驱动程序提供高层次的 API，几种常用 USB 设备类的特点如下：

① 使用极为方便。即设备设置包括创建一组静态的数据结构和调用单一的初始化 API。

② 可配置的 VID/PID、电源参数和字符串表、使设备易于定制，而无需修改任何库函数代码。

③ 一致的接口。即所有的设备类驱动程序库使用类似的 API，使其非常简捷地在它们之间移动。

④ 最小的应用程序开销。即绝大多数的 USB 处理都是在类驱动程序中进行的，低层只用于处理应用程序中的数据读/写。

⑤ 可选配的 USB 缓冲区对象可用来进一步简化数据发送和接收。即使用 USB 缓冲区，与设备类驱动程序的互动可变为一个简单的读/写 API，而无需状态机来确保在正确的时间发送或接收数据。

⑥ 设备类驱动程序的 API 完全封装了底层 USB 设备和 USB 驱动程序的 API，使应用程序只有一个单一的 API 接口。

折中这些优势，要求应用程序开发人员在使用设备类驱动程序库 API 时应注意以下限制：

① 当设备类驱动程序 API 在使用时，不可调用其他任何的 USB 层。

② 设备类驱动程序不支持切换配置。

注意：目前提供的设备类驱动程序，允许创建一个通用的大容量设备、一个通信设备类设备（虚拟串口）和人机接口设备类设备（鼠标、键盘、游戏杆等），也包括用于复合设备的特殊类驱动程序。作为一个包装，允许在一台设备中使用多个设备类驱动程序。

4）USB 设备类 API（USB Device Class API）：在某些情况下，使用标准的设备类可创建多种不同和多样的设备，在该状况下，一个额外的 API 层将为专门的设备提供进一步的操作和简化应用程序的接口。例如，人机接口设备（HID）类就是这样的一个类，用于支持多种设备，包括键盘、游戏杆、鼠标和游戏控制器等。HID 设备类驱动程序是非常通用的，可支持尽可能宽泛的设备。为了简化使用的接口，它提供特定的 API 来支持 BIOS 兼容的键盘和鼠标操作，使用鼠标类 API 替代基本的 HID 类驱动程序 API，因而，应用程序可使自己对于 USB 主机使用非常简单的接口作为一个鼠标可见，包括初始化调用和鼠标移动或按钮按下的通知调用等。在使用键盘设备类的 API 时，应用程序可使用单一 API 发送按键以及间断信息到主机而无需知道底层结构和 USB HID 协议，因而可大大简化 USB 设备的编程及操作。

14. 3. 2 Driverlib 库函数介绍

USB 固件库提供一组用于访问 Tiva USB 设备、主机和/或设备、或 OTG 控制器的 API 函数。根据 USB 控制器在微控制器中提供的功能，可将 API 函数分成几组：USBDev、USB-Host、USBOTG、USBEndpoint 和 USBFIFO。具有一个 USB 器件控制器的微控制器只可使用 USBDev 组的 API；仅含一个 USB 主机控制器的微控制器只能使用 USBHOST 组中的 API；仅含一个 OTG 接口的微控制器使用 USB OTG 组的 API。对于 USB OTG 控制器，一旦配置了 USB 控制器的模式，将使用设备或主机的 API。其余部分的 APIs 既可用于 USB 主机控制器，也可用于 USB 设备控制器。USBEndpoint API 用于配置和访问端点，而 USBFIFO API 用于配置 FIFOs 的大小和位置。也就是说，采用 Driverlib 固件库函数即可搭建 14. 1 节介绍的 USB 2. 0 的内容。该驱动程序包含在 Driverlib/usb. c 中，Driverlib/usb. h 定义了 USB API 函数的定义。

USB 控制器 API 根据使用条件在 driverlib 库手册中的归类如图 14-10 所示。

下面仅就实现 USB 基本功能的 API 函数集“通用 USB API 函数”和“带有 uDMA 控制器的 USB 操作”的功能作一简单介绍，而对于 USB 锦上添花的 API 函数集“带有 DMA 控制器的 USB 操作”，“USB 连接电源管理”与“ULPI”所包含的 API 功能简介，请参考 Driverlib 库的 USB 部分。有关 USB 驱动库的函数功能说明，请参考书后第 14 章附录。

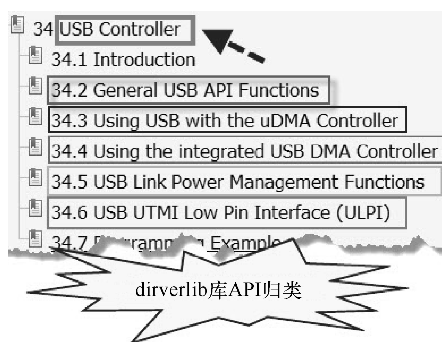


图 14-10　USB API 在手册中的归类

（1）通用 USB API 函数

它包括 70 个用于底层/中层的 API 函数，提供了实现 USB 器件或 USB 主机堆栈所需的全部函数。基于使用中的 USB 控制器类型 API 可抽象出 IN/OUT 端点的特征。使用 IN/OUT 术语的任何 API 函数符合这些条款的标准 USB 描述。例如，对于一个只具有设备接口的微控制器上的 OUT 端点，实际上它会通过该端点来接收数据，而对于有一个主机接口的微控制器，实际上它也将在 OUT 端点上发送数据。

USB 控制器具有一个可分配给端点的全局 FIFO 存储器空间。FIFO RAM 总的空间大小为 2048 字节或 4096 字节，这取决于所使用的 Tiva 设备。要特别注意的是：存储器中的前 64 个字节专门用于控制传输的端点 0。剩余的 1984 或 4032 个字节可根据应用程序的需要来

配置。通常在应用的开始进行 FIFO 的配置，而一旦进入使用就不可对其配置进行修改。可通过 USBFIFOConfig( ) API 函数来配置每一个端点的 FIFO 起始地址和大小。

（2）带有 uDMA 控制器的 USB 操作

USB 控制器可以和 uDMA 一起用于主机/设备控制器的数据接收与发送。uDMA 控制器不可访问端点 0，然而，所有其他端点都可使用 UDMA 控制器访问。USB 的 uDMA 通道号由下列值定义：

- DMA_CHANNEL_USBEP1RX。
- DMA_CHANNEL_USBEP1TX。
- DMA_CHANNEL_USBEP2RX。
- DMA_CHANNEL_USBEP2TX。
- DMA_CHANNEL_USBEP3RX。
- DMA_CHANNEL_USBEP3TX。

对于超过 8 个端点设备，必须使用 USBEndpointDMAChannel( ) 函数把 3 个 DMA 发送通道和 DMA 接收通道中的一个分配给所需的端点。由于 uDMA 控制器把传输看作发送或接收，并且 USB 控制器对 IN/OUT 事务进行操作，则需小心使用正确的 uDMA 通道与正确的端点。USB 主机 IN 端点和 USB 设备 OUT 端点都可以使用接收 uDMA 通道，而 USB 主机 OUT 端点和 USB 设备 IN 端点可使用 UDMA 发送通道。

当配置端点时，需另外的 DMA 设置。当调用 USBDevEndpointConfig( ) 函数来配置一个端点时需使用 uDMA，额外的标志必须被添加到 ulFlags 参数。这些标志是 USB_EP_DMA_MODE_0 或 USB_EP_DMA_MODE_1 其中之一，以控制 DMA 事务的模式，一旦包准备就绪可使用 USB_EP_AUTO_SET 允许数据自动传输。在使用 USB_EP_DMA_MODE_0，当全部传输完成时，USB 控制器只产生一个中断。因此，应用程序在配置 DMA 传输之前必须知道完整传输的大小。在 USB_EP_DMA_MODE_1，只有当数据包传输完成和短包中断时，USB 控制器才会发出 DMA 请求。短包数据仍保留在 USB FIFO，因此必须启动 FIFO 中数据的另一次传输。

下面是一个有关带 uDMA 的端点配置的例子。

（1）设备 IN 端点的配置

```
//
//端点 1 是使用 DMA 的一个设备模式的 BULK IN 端点
//
USBDevEndpointConfigSet(USB0_BASE,USB_EP_1,64,
                        (USB_EP_MODE_BULK | USB_EP_DEV_IN |
                         USB_EP_DMA_MODE_0 | USB_EP_AUTO_SET));
```

（2）将端点 1 配置为发送通道

```
//
//将 DMA 设置为 USB 发送
//
DMAChannelAttributeClear(DMA_CHANNEL_USBEP1TX,DMA_CONFIG_ALL);
//
//使能 uDMA 突发模式
//
```

```
DMAChannelAttributeSet(DMA_CHANNEL_USBEP1TX,DMA_CONFIG_USEBURST);
//
//数据长度为 8 位且源中有一个字节的增量
//目标作为一个 FIFO 没有增量
//
DMAChannelControlSet(DMA_CHANNEL_USBEP1TX,DMA_DATA_SIZE_8,DMA_ADDR_INC_8,
DMA_ADDR_INC_NONE,DMA_ARB_64,0);
```

(3) 启动端点 1 的数据传输

```
//
//设置要传输数据的地址和长度
//
DMAChannelTransferSet(DMA_CHANNEL_USBEP1TX,DMA_MODE_BASIC,pData,
                      USBFIFOAddr(USB0_BASE,USB_EP_1),64);
//
//启动传输
//
DMAChannelEnable(DMA_CHANNEL_USBEP1TX);
```

(4) 中断处理

```
if((g_ulFlags & EP1_DMA_IN_PEND)&&
    (DMAChannelModeGet(DMA_CHANNEL_USBEP1TX) == DMA_MODE_STOP))
{
//
//处理 DMA 完成情况
//
...
}
//
//获取中断状态
//
ulStatus = USBIntStatusEndpoint(USB0_BASE);
if(ulStatus & USB_INTEP_DEV_IN_1)
{
//
//处理程序的传输完成情况
//
...
}
```

(5) 将端点 1 配置成接收通道

```
//
//端点 1 是一个使用 DMA 的设备模式 BULK OUT 端点
//
USBDevEndpointConfigSet(USB0_BASE,USB_EP_1,64,
                        (USB_EP_DEV_OUT | USB_EP_MODE_BULK |
                         USB_EP_DMA_MODE_1 | USB_EP_AUTO_CLEAR));
```

(6) 配置端点 1 发送通道

```
//
//清除任何 uDMA 设置
//
DMAChannelAttributeClear(DMA_CHANNEL_USBEP1RX,DMA_CONFIG_ALL);
DMAChannelAttributeSet(DMA_CHANNEL_USBEP1RX,DMA_CONFIG_USEBURST);
DMAChannelControlSet(DMA_CHANNEL_USBEP1RX,DMA_DATA_SIZE_8,
                     DMA_ADDR_INC_NONE,DMA_ADDR_INC_8,DMA_ARB_64,0);
```

(7) 启动端点 1 的数据请求

```
//
//配置要传输数据的地址和长度。传输从 USB FIFO 端点 0 到 g_DataBufferIn
//
DMAChannelTransferSet(DMA_CHANNEL_USBEP1RX,DMA_MODE_BASIC,
                      USBFIFOAddr(USB0_BASE,USB_EP_1),g_DataBufferIn,
                      64);
//
//使能 uDMA 通道,并等待数据
//
DMAChannelEnable(DMA_CHANNEL_USBEP1RX);
```

(8) 带短包的中断处理

```
//
//获取当前的中断状态
//
ulStatus = USBIntStatusEndpoint(USB0_BASE);
if(ulStatus & USB_INTEP_DEV_OUT_1)
{
//
//短包处理
//
...
}
else if((g_ulFlags & EP1_DMA_OUT_PEND)&&
(DMAChannelModeGet(DMA_CHANNEL_USBEP1RX) == DMA_MODE_STOP)
{
//
//处理 DMA 完成情况
//
...
//
//如果需要重启 DMA 接收
//
...
}
```

下面是 Driverlib 库例程片段:

```
// ***********************************************************************
//实现端点配置与数据传输
// ***********************************************************************

//
//将端点 1 配置成 Bulk IN 端点,最大包长度为 64 个字节
//
USBDevEndpointConfigSet(USB0_BASE,USB_EP_1,64,       //最大包长度
                        DISABLE_NAK_LIMIT,
                        USB_EP_MODE_BULK |           //批量(Bulk)模式
                        USB_EP_DEV_IN);              //设备模式的 IN 端点
//
//将 FIFO 配置成起始地址为 64 的设备 IN 端点,并且它的包长度为 64 B
//
USBFIFOConfig(USB0_BASE,USB_EP_1,64,USB_FIFO_SZ_64,USB_EP_DEV_IN);
...
//
//将数据存放到 FIFO 中
//
USBEndpointDataPut(USB0_BASE,USB_EP_1,pucData,64);
//
//启动数据的传输
//
USBEndpointDataSend(USB0_BASE,USB_EP_1,USB_TRANS_IN);
```

14.3.3 USBlib 库函数介绍

德州仪器 USB 库用于创建 USB 设备、主机或 OTG 应用程序的一组数据类型和函数。它提供了多个编程接口，从仅提取底层 USB 控制器硬件的最薄层到为特定器件提供简单 API 支持的高级接口。USB 库的内容和相关的头文件分为四个主要组：

① 通用函数：可用于设备、主机和双模式应用程序中。包含解析 USB 描述符和 USB 库的配置特点。

② 设备模式的特定函数：提供所有 USB 器件应用程序需要的类独立功能，例如主机连接信令和对标准描述符请求的响应。

③ 主机模式的特定函数：提供所有 USB 主机应用程序需要的类独立功能，例如器件检测和枚举以及端点管理。

④ 模式检测与控制函数：用于 USB 的操作模式检测，如检测 USB 设备间的接入，以及监测 VBUS 和 ID 引脚信号等。

此外，库中还提供驱动程序与“.inf”文件，这将有助于开发人员为新设备定制驱动程序和“.inf”文件。

1. USBlib 目录结构概述

TivaWare 2.0 版软件的 USBlib 目录结构如图 14-11 所示。

2. 通用 API

下面介绍 USB 库中通用 API 的数据类型和函数，不包括特定于 USB 主机、设备或 OTG 操作。通用 API 的函数和类型描述可分为三大类：

图 14-11　USB 库的目录结构

1）USB 标准类型定义（USB2.0 规范第 9 章）和函数解析。

2）主机/设备模式切换为 OTG 或双模式应用。

3）USB 设备类头文件。

通用 API 的源代码（如图 14-11 所示）如下：

1）usblib.h：包含 USB 库中所有数据类型和定义（将库作为一个整体）的主头文件。并且包含通用 API 函数的原型以及 USB 库的所有客户端应用程序都应包含该头文件。

2）usb-ids.h：包含评估套件中每个 USB 例子的德州仪器的厂商号（VID）和产品号（PID）的头文件。

3）usbmsc.h：包含 USB 大容量存储类定义的头文件。

4）usbcdc.h：包含 USB 通信设备类定义的头文件。

5）usbhid.h：包含 USB 人机接口设备类定义的头文件。

6）usblibpriv.h：用于在 USB 文库的各种组件中共享变量和定义的私有头文件，客户端应用程序不应包括这个头文件。

7）usbdesc.c：解析 USB 描述符函数的源代码。

8）usbbuffer.c：用于支持一些应用中 USB 端点数据流缓冲区的一组函数源代码。

9）usbringbuf.c：实现简单环缓冲区的一组函数源代码。

10）usbmode.c：用于在主机和设备操作之间切换的一组函数源代码。

11）usbtick.c：内部 USB 库滴答时钟处理函数的源代码。此文件不包含由应用程序访问的任何功能。

3. 主机 API

下面介绍在主机模式下由 USB 库提供的对 USB 控制器的支持。为了简化应用程序与添加新的设备和设备类，USB 库为 USB 主机控制器提供了一个分层接口。在 USB 库的顶层提供了由 USB 库支持的方便访问各类外设的应用程序接口。下面这个层是处理每种类型设备细节的 USB 主机控制器的设备接口，以及如何与 USB 主机类驱动程序进行通信。USB 主机类驱动程序是处理整个设备类的基础，如 HID 和大容量存储类设备。USB 主机类驱动程序层与最底层的 USB 主控制器驱动程序库通信。通过直接访问最底层 DriverLib 中的函数来提供与已连接的 USB 设备进行通信。

主机接口的分层如下：

1）Device APIs（鼠标、键盘、文件系统）。

2）USB Class Driver APIs（HID、大容量存储、HUB）。

3）USB Host Controller APIs。

4）DriverLib USB Driver APIs。

（1）主机 API 的源代码（如图 14-12 所示）

1） usbhost. h　　　　//在 USB 库中包含主机模式的函数原型和数据类型定义的头文件

2） usbhaudio. h　　　//包含主机支持的特定音频类定义的头文件

3） usbhhid. h　　　　//包含与 USB 主机 HID 类驱动程序交互所需定义的头文件

4） usbhhub. h　　　　//包含与 USB 主机 Hub 类驱动程序交互所需定义的头文件

5） usbhhidkeyboard. h　//包含与 USB 主机 HID 类键盘设备交互所需定义的头文件

6） usbhhidmouse. h　　//包含与 USB 主机 HID 类鼠标设备交互所需定义的头文件

7） usbhmsc. h　　　　//包含主机支持的特定大容量存储类定义的头文件

8） usbhostenum. c　　//由库提供的 USB 主机枚举函数的源代码

9） usbhaudio. c　　　//USB 主机音频类驱动程序的源代码

10） usbhhid. c　　　　//USB 主机 HID 类驱动程序的源代码

11） usbhhub. c　　　　//USB 主机集线器（HUB）类驱动程序的源代码

12） usbhhidkeyboard. c　//USB 主机 HID 类键盘设备的源代码

13） usbhhidmouse. c　　//USB 主机 HID 类的鼠标设备的源代码

14） usbhmsc. c　　　　//USB 主机大容量存储类驱动程序的源代码

15） usbhscsi. c　　　　//调用主机大容量存储类驱动程序的高级 SCSI 接口的源代码

（2）主机专用类及定义（如图 14-13 所示）

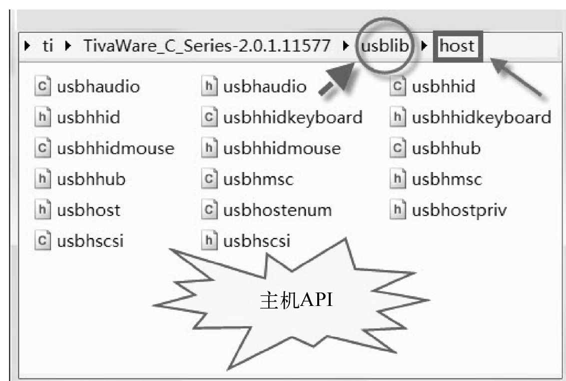


图 14-12　主机 API 的源代码



图 14-13　主机专用类及定义

由于 USBlib 固件库的手册多达 300 页，这里就不一一介绍了，需要这部分知识的读者可以参考 USBlib 库手册图示的内容。而具体用法可参考 DK－TM4C123G 开发板提供的例程：

1）usb_host_audio。

2）usb_host_keyboard。

3）usb_host_mouse。

4）usb_host_msc。

4. 设备 API 函数

下面介绍 USB 库中提供支持 USB 设备应用程序的各种 API 层。一些编程接口提供了从仅抽象底层 USB 控制器硬件的最薄层到提供简单 API 支持的特定设备接口的高层。

USB 库包含了与开发 USB 设备应用程序相关的四个 API 层。从上往下向堆栈移动，虽然每个 API 层给编程 USB 应用程序提供了更大的灵活性，但使用较低层需要耗费更多的时间来实现同样的功能。与 USB 库的主机编程接口类似，设备接口的分层如下：

1）Device Class APIs。

2）Device Class Driver APIs。

3）The USB Device API。

4）The USB DriverLib API。

（1）设备 API 的源代码（如图 14-14 所示）

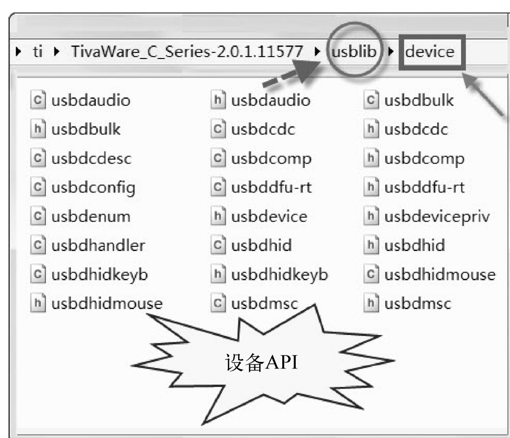


图 14-14　设备 API 的源代码

1）usbdbulk. h　　//定义 USB 通用批量设备类驱动程序 API 的头文件

2）usbdcdc. h　　//定义了 USB 通信设备类（CDC）设备类驱动程序 API 的头文件

3）usbdhid. h　　//定义 USB 人机接口设备（HID）设备类驱动程序 API 的头文件

4）usbdhidkeyb. h　　//定义 USB HID 键盘设备类 API 的头文件

5）usbdhidmouse. h　　//定义 USB HID 鼠标设备类 API 的头文件

6）usbdevicepriv. h　　//用于在 USB 文库的各种组件中共享变量和定义的私有头文件

//客户端应用程序不应包括这个头文件

7） usbdenum. c　　//由库提供的 USB 设备枚举函数的源代码

8） usbdhandler. c　　//USB 设备中断处理程序的源代码

9） usbdconfig. c　　//USB 设备配置函数的源代码

10） usbdcdesc. c　　//解析配置描述符定义函数的源代码（与 USB 设备 API 一起使用）

11） usbdbulk. c　　//USB 通用批量设备类驱动程序的源代码

12） usbdcdc. c　　//USB 通信设备类（CDC）设备类驱动程序的源代码

13） usbdhid. c　　//USB 人机接口设备（HID）设备类驱动程序的源代码

14） usbdhidkeyb. c　//USB HID 键盘设备类的源代码

15） usbdhidmouse. c //USB HID 鼠标设备类的源代码

（2）设备专用类及定义（如图 14-15 所示）

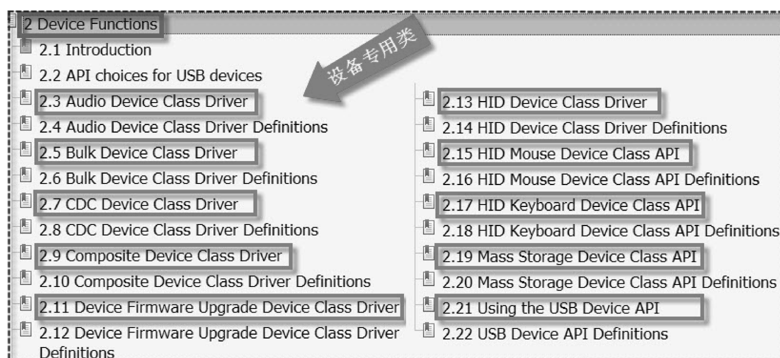


图 14-15　设备专用类及定义

图 14-15 的内容较多，这里就不一一介绍了，需要用到这部分知识的读者可以参考 USBlib 库手册的图示部分。而具体 USB 设备 API 的用法可参考 TI 提供的例程，比如 EK-TM4C123GXL（LaunchPad）开发板提供了 usb_dev_bulk 和 usb_dev_serial 两个有关 USB 设备的例程，DK-TM4C123G 开发板除这两个例程外还提供了 usb_dev_keyboard 和 usb_dev_msc 两个 USB 设备例程。下面将以 usb_dev_bulk 例程来介绍设备专用库、描述符、底层驱动等的用法。

注意：在开发基于 USB 的应用时，原则上尽量采用 TI USBlib 库提供的专用类 API 来实现，对目前还无法实现的功能，可以使用较低层的现有固件库函数来定制。



14.4　例程

本小节通过一个 TI 提供的 usb_bulk_exmaple 例程来介绍基于 USB 应用的程序设计及测试。

1. USB 硬件连接

USB 硬件连接如图 14-16 所示。

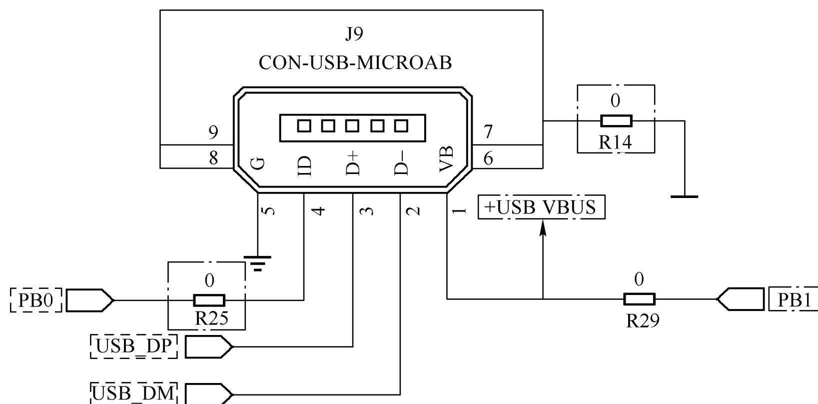


图 14-16　LaunchPad 开发板的 USB 硬件接线图

2. usb_bulk 程序及测试

（1）usb_bulk_structs. c 说明

```
// ****************************************************************************
//文件名: usb_bulk_structs. c
//来源:TI 例程
//功能:定义批量 USB 设备的数据结构
//程序框架,如图 14-17 所示
/ *
```

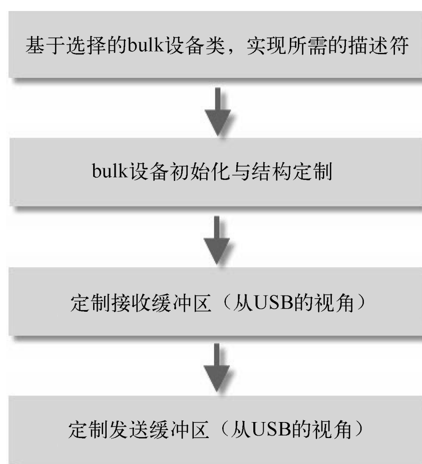


图 14-17　usb_bulk_structs. c 程序框图

```
 * /
//Tiva 2. 0 版 EK - TM4C123GXL 固件包例程
// ****************************************************************************

#include < stdint. h >
#include < stdbool. h >
#include " inc/hw_types. h"
#include " driverlib/usb. h"
#include " usblib/usblib. h"
#include " usblib/usb - ids. h"
#include " usblib/device/usbdevice. h"
#include " usblib/device/usbdbulk. h"
#include " usb_bulk_structs. h"

// ****************************************************************************
//设备支持的语言
// ****************************************************************************
const uint8_t g_pui8LangDescriptor[ ] =
{
    4,
    USB_DTYPE_STRING,
    USBShort( USB_LANG_EN_US)
};

// ****************************************************************************
```

```
//制造商字符串
//***************************************************************************
const uint8_t g_pui8ManufacturerString[ ] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};

//***************************************************************************
//产品字符串
//***************************************************************************
const uint8_t g_pui8ProductString[ ] =
{
    (19 + 1) * 2,
    USB_DTYPE_STRING,
    'G', 0, 'e', 0, 'n', 0, 'e', 0, 'r', 0, 'i', 0, 'c', 0, ' ', 0, 'B', 0,
    'u', 0, 'l', 0, 'k', 0, ' ', 0, 'D', 0, 'e', 0, 'v', 0, 'i', 0, 'c', 0,
    'e', 0
};

//***************************************************************************
//序列号字符串
//***************************************************************************
const uint8_t g_pui8SerialNumberString[ ] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

//***************************************************************************
//数据接口描述字符串
//***************************************************************************
const uint8_t g_pui8DataInterfaceString[ ] =
{
    (19 + 1) * 2,
    USB_DTYPE_STRING,
    'B', 0, 'u', 0, 'l', 0, 'k', 0, ' ', 0, 'D', 0, 'a', 0, 't', 0,
    'a', 0, ' ', 0, 'I', 0, 'n', 0, 't', 0, 'e', 0, 'r', 0, 'f', 0,
    'a', 0, 'c', 0, 'e', 0
};

//***************************************************************************
//配置描述字符串
//***************************************************************************
const uint8_t g_pui8ConfigString[ ] =
{
    (23 + 1) * 2,
```

```
    USB_DTYPE_STRING,
    'B',0,'u',0,'l',0,'k',0,' ',0,'D',0,'a',0,'t',0,
    'a',0,' ',0,'C',0,'o',0,'n',0,'f',0,'i',0,'g',0,
    'u',0,'r',0,'a',0,'t',0,'i',0,'o',0,'n',0
};

//*****************************************************************************
//描述符的字符串表
//*****************************************************************************
const uint8_t * const g_ppui8StringDescriptors[] =
{
    g_pui8LangDescriptor,
    g_pui8ManufacturerString,
    g_pui8ProductString,
    g_pui8SerialNumberString,
    g_pui8DataInterfaceString,
    g_pui8ConfigString
};

#define NUM_STRING_DESCRIPTORS (sizeof(g_ppui8StringDescriptors)/          \
                                sizeof(uint8_t *))

//*****************************************************************************
//
//bulk 设备初始化与结构定制。因此,可在 bulk 设备类驱动程序和应用程序代码之间使用 USBBuffers
//函数指针和回调数据值被设置为插入到每一个数据信道,发送和接收缓冲区中。在恰当的缓冲
//区中,bulk 通道的 callback 被设置成相关的 通道函数,以及回调数据被设置成指向通道的实例
//数据。反过来,缓冲区的 callback 又设置为应用程序的函数和回调数据设置成该 bulk 实例结构
//*****************************************************************************
extern const tUSBBuffer g_sTxBuffer;
extern const tUSBBuffer g_sRxBuffer;

tUSBDBulkDevice g_sBulkDevice =
{
    USB_VID_TI_1CBE,
    USB_PID_BULK,
    500,
    USB_CONF_ATTR_SELF_PWR,
    USBBufferEventCallback,
    (void *)&g_sRxBuffer,
    USBBufferEventCallback,
    (void *)&g_sTxBuffer,
    g_ppui8StringDescriptors,
    NUM_STRING_DESCRIPTORS
};

//*****************************************************************************
//接收缓冲区(从 USB 的视角)
//*****************************************************************************
uint8_t g_pui8USBRxBuffer[BULK_BUFFER_SIZE];
```

```
uint8_t g_pui8RxBufferWorkspace[USB_BUFFER_WORKSPACE_SIZE];
const tUSBBuffer g_sRxBuffer =
{
    false,                          //这是一个接收缓冲区
    RxHandler,                      //pfnCallback
    (void *)&g_sBulkDevice,         //回调数据是设备指针
    USBDBulkPacketRead,             //pfnTransfer
    USBDBulkRxPacketAvailable,      //pfnAvailable
    (void *)&g_sBulkDevice,         //pvHandle
    g_pui8USBRxBuffer,              //pi8Buffer
    BULK_BUFFER_SIZE,               //ui32BufferSize
    g_pui8RxBufferWorkspace         //pvWorkspace
};

//*****************************************************************************
//发送缓冲区(从 USB 的视角)
//*****************************************************************************
uint8_t g_pui8USBTxBuffer[BULK_BUFFER_SIZE];
uint8_t g_pui8TxBufferWorkspace[USB_BUFFER_WORKSPACE_SIZE];
const tUSBBuffer g_sTxBuffer =
{
    true,                           //这是一个发送缓冲区
    TxHandler,                      //pfnCallback
    (void *)&g_sBulkDevice,         //回调数据是设备指针
    USBDBulkPacketWrite,            //pfnTransfer
    USBDBulkTxPacketAvailable,      //pfnAvailable
    (void *)&g_sBulkDevice,         //pvHandle
    g_pui8USBTxBuffer,              //pi8Buffer
    BULK_BUFFER_SIZE,               //ui32BufferSize
    g_pui8TxBufferWorkspace         //pvWorkspace
};
```

(2) usb_dev_bulk.c 介绍

```
//*****************************************************************************
//文件名:usb_dev_bulk.c - Main routines for the generic bulk device example.
//来源:TI 例程
//功能:
//!这个例子提供一个通用的 USB 设备实现与主机简单的 bulk 数据传输
//!该设备使用供应商特定的类 ID,并支持单一 bulk IN 端点和单一 bulk OUT 端点
//!从主机接收到的数据被假定为 ASCII 文本,并以大小写颠倒的方式回显所有字母
//!设备的 Windows INF 文件在 C:/TI/TivaWare C 系列/windows_drivers 目录下。这个 INF 包含在
//! Windows XP 和 Vista 的 PC 上安装 WinUSB 子系统所需的信息。WinUSB 是 Windows 子系统,
//!其允许在用户模式下应用程序访问 USB 设备而不需要供应商特定的内核模式驱动程序。这个
//!Windows 命令行应用程序例程(usb_bulk_example)展示了如何与 bulk 设备连接与通信
//!需安装 USB Windows 端例子包"SW-USB-WIN",可在 http://www.ti.com/tivaware 网址下载
//!使用微软的 VisualStudio 2008 包含的工程文件就可以构建这个例程。此应用程序的源代码可
//!以在目录 TivaWare-for-C-Series/tools/usb_bulk_example 中找到
/*
```

USB 通用 Bulk 设备模型如图 14-18 所示。

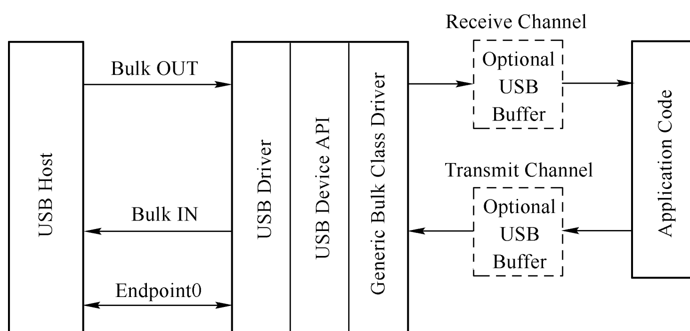


图 14-18　USB 通用 Bulk 设备模型

```
*/
//************************************************************************
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/fpu.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/systick.h"
#include "driverlib/timer.h"
#include "driverlib/uart.h"
#include "driverlib/rom.h"
#include "usblib/usblib.h"
#include "usblib/usb-ids.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdbulk.h"
#include "utils/uartstdio.h"
#include "utils/ustdlib.h"
#include "usb_bulk_structs.h"
//************************************************************************
//定义系统时钟速率表示每秒的滴答数和毫秒周期
//************************************************************************
#define SYSTICKS_PER_SECOND      100
#define SYSTICK_PERIOD_MS        (1000/SYSTICKS_PER_SECOND)
//************************************************************************
//全局系统时钟计数器
//************************************************************************
volatile uint32_t g_ui32SysTickCount = 0;
//************************************************************************
//跟踪发送和接收计数值
//************************************************************************
volatile uint32_t g_ui32TxCount = 0;
```

```
volatile uint32_t g_ui32RxCount = 0;
#ifdef DEBUG
uint32_t g_ui32UARTRxErrors = 0;
#endif
// *********************************************************************
//与调试相关的定义和声明
//如果在调试过程中定义了 DEBUG,可通过 UART0 端口输出调试结果
// *********************************************************************
#ifdef DEBUG
// *********************************************************************
//映射所有的调试打印可调用 UARTprintf 调试版本
// *********************************************************************
#define DEBUG_PRINT UARTprintf
#else
// *********************************************************************
//编译所有调试打印调用发布版本
// *********************************************************************
#define DEBUG_PRINT while(0)((int (*)(char *,...))0)
#endif
// *********************************************************************
//用于从中断上下到主循环文命令传递的标志
// *********************************************************************
#define COMMAND_PACKET_RECEIVED   0x00000001
#define COMMAND_STATUS_UPDATE     0x00000002
volatile uint32_t g_ui32Flags = 0;
// *********************************************************************
//指示已配置 USB 的全局标志
// *********************************************************************
static volatile bool g_bUSBConfigured = false;
// *********************************************************************
//当驱动程序库遇到错误时,调用错误处理程序
// *********************************************************************
#ifdef DEBUG
void
__error__(char *pcFilename,uint32_t ui32Line)
{
    UARTprintf("Error at line %d of %s\n",ui32Line,pcFilename);
    while(1)
    {
    }
}
#endif
// *********************************************************************
//系统时钟计数器的中断处理程序
// *********************************************************************
void
SysTickIntHandler(void)
{
    //更新系统时钟计数器
```

```
    g_ui32SysTickCount ++ ;
}
// ************************************************************************
//接收新数据并将其返回给主机
//\参数 psDevice 指向设备要处理数据的数据实例
//\参数 pui8Data 指向 USB 接收缓冲区新接收的数据
//\参数 ui32NumBytes 为待处理数据的字节数
//当收到可从主机获取数据通知,调用此函数
//逐字节读出数据,将发现的任何字母字符大小写交换,然后把它写回到主机
//\ return 返回处理后的数据字节数
// ************************************************************************
static uint32_t
EchoNewDataToHost(tUSBDBulkDevice *psDevice,uint8_t *pui8Data,
                  uint32_t ui32NumBytes)
{
    uint32_t ui32Loop,ui32Space,ui32Count;
    uint32_t ui32ReadIndex;
    uint32_t ui32WriteIndex;
    tUSBRingBufObject sTxRing;
    //
    //获取当前缓冲区信息,以使可直接向发送缓冲区写入数据
    //
    USBBufferInfoGet(&g_sTxBuffer,&sTxRing);
    //
    //发送缓冲区中有多少空间?
    //
    ui32Space = USBBufferSpaceAvailable(&g_sTxBuffer);
    //
    //这一次有多少个字符待处理?
    //
    ui32Loop = (ui32Space < ui32NumBytes) ? ui32Space: ui32NumBytes;
    ui32Count = ui32Loop;
    //
    //更新接收计数器
    //
    g_ui32RxCount += ui32NumBytes;
    //
    //转存调试信息
    //
    DEBUG_PRINT("Received %d bytes\n",ui32NumBytes);
    //
    //通过直接访问 USB 缓冲区来设置要处理的字符
    //
    ui32ReadIndex = (uint32_t)(pui8Data - g_pui8USBRxBuffer);
    ui32WriteIndex = sTxRing. ui32WriteIndex;
    while(ui32Loop)
    {
        //
        //将转换字符从接收缓冲区复制到发送缓冲区
```

```
        //
        //这是一个小写字符?
        //
        if((g_pui8USBRxBuffer[ui32ReadIndex] >= 'a') &&
          (g_pui8USBRxBuffer[ui32ReadIndex] <= 'z'))
        {
            //
            //转换为大写字母并将其写入到发送缓冲区
            //
            g_pui8USBTxBuffer[ui32WriteIndex] =
                (g_pui8USBRxBuffer[ui32ReadIndex] - 'a') + 'A';
        }
        else
        {
            //
            //这是一个大写字符?
            //
            if((g_pui8USBRxBuffer[ui32ReadIndex] >= 'A') &&
              (g_pui8USBRxBuffer[ui32ReadIndex] <= 'Z'))
            {
                //
                //转换为小写并将其写入到发送缓冲区
                //
                g_pui8USBTxBuffer[ui32WriteIndex] =
                    (g_pui8USBRxBuffer[ui32ReadIndex] - 'Z') + 'z';
            }
            else
            {
                //
                //将接收到的字符复制到发送缓冲区
                //
                g_pui8USBTxBuffer[ui32WriteIndex] =
                        g_pui8USBRxBuffer[ui32ReadIndex];
            }
        }
        //
        //移动到下一个字符,如果需要,注意调整缓冲区的指针
        //
        ui32WriteIndex++;
        ui32WriteIndex = (ui32WriteIndex == BULK_BUFFER_SIZE)?
                        0: ui32WriteIndex;
        ui32ReadIndex++;
        ui32ReadIndex = (ui32ReadIndex == BULK_BUFFER_SIZE)?
                        0: ui32ReadIndex;
        ui32Loop--;
    }
    //
    //完成处理准备就绪的数据后,立即将其传回主机
    //
```

```
    USBBufferDataWritten( &g_sTxBuffer,ui32Count);
    DEBUG_PRINT("Wrote %d bytes\n",ui32Count);
    //
    //为了尽可能多地处理直接来自接收缓冲区中的数据(需要返回的字节数),允许下层适当
    //地更新其读出指针
    //
    return(ui32Count);
}
//*****************************************************************************
//处理 bulk 驱动器通知相关的传输信道(数据到 USB 主机)
//
//\参数 pvCBData 是该通道客户端提供的回调(callback)指针
//\参数 ui32Event 确定要通知的事件
//\参数 ui32MsgValue 是特定事件的值
//\参数 pvMsgData 是特定事件的指针
//
//该函数被 bulk 驱动器调用来通知与传输数据通道相关操作的任何事件(在 IN 通道传送数据到
//USB 主机)
//
//\ return 返回值是特定事件
//*****************************************************************************
uint32_t
TxHandler(void *pvCBData,uint32_t ui32Event,uint32_t ui32MsgValue,
          void *pvMsgData)
{
    //
    //在该例子中不需要响应任何发送事件,仅对发送计数器更新即可
    //
    if(ui32Event == USB_EVENT_TX_COMPLETE)
    {
        g_ui32TxCount += ui32MsgValue;
    }
    //
    //转存调试信息
    //
    DEBUG_PRINT("TX complete %d\n",ui32MsgValue);
    return(0);
}
//*****************************************************************************
//处理 bulk 驱动器通知相关接收通道(数据来自 USB 主机)
//
//\参数 pvCBData 是该通道客户端提供的回调指针
//\参数 ui32Event 确定要通知的事件
//\参数 ui32MsgValue 是特定事件值
//\参数 pvMsgData 是特定事件的指针
//
//该函数被 bulk 驱动器调用来通知与接收数据通道相关操作的任何事件(在 OUT 通道传送来自
//USB 主机的数据)
//
```

```
//\ return 返回值是特定事件
// *******************************************************************
uint32_t
RxHandler(void  * pvCBData,uint32_t ui32Event,
               uint32_t ui32MsgValue,void  * pvMsgData)
{
    //
    //哪个事件正在发送?
    //
    switch(ui32Event)
    {
        //
        //连接到一个主机并可能马上进行通信
        //
        case USB_EVENT_CONNECTED:
        {
            g_bUSBConfigured = true;
            UARTprintf("Host connected. \n");
            //
            //刷新缓冲区
            //
            USBBufferFlush(&g_sTxBuffer);
            USBBufferFlush(&g_sRxBuffer);
            break;
        }
        //
        //断开主机连接
        //
        case USB_EVENT_DISCONNECTED:
        {
            g_bUSBConfigured = false;
            UARTprintf("Host disconnected. \n");
            break;
        }
        //
        //已经收到一个新的数据包
        //
        case USB_EVENT_RX_AVAILABLE:
        {
            tUSBDBulkDevice  * psDevice;
            //
            //从回调数据参数获得一个指向数据实例的指针
            //
            psDevice = (tUSBDBulkDevice  * )pvCBData;
            //
            //读取新的数据包并返回到主机
            //
            return(EchoNewDataToHost(psDevice,pvMsgData,ui32MsgValue));
        }
```

```
        //
        //忽略挂起和暂时恢复
        //
        case USB_EVENT_SUSPEND:
        case USB_EVENT_RESUME:
        {
            break;
        }
        //
        //忽略所有其他事件并返回 0
        //
        default:
        {
            break;
        }
    }
    return(0);
}
//*****************************************************************************
//配置 UART 和引脚
//*****************************************************************************
void
ConfigureUART(void)
{
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    ROM_GPIOPinConfigure(GPIO_PA0_U0RX);
    ROM_GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE,GPIO_PIN_0 | GPIO_PIN_1);
    UARTClockSourceSet(UART0_BASE,UART_CLOCK_PIOSC);
    UARTStdioConfig(0,115200,16000000);
}
//*****************************************************************************
//主应用程序的入口函数
//*****************************************************************************
int
main(void)
{
    volatile uint32_t ui32Loop;
    uint32_t ui32TxCount;
    uint32_t ui32RxCount;
    ROM_FPULazyStackingEnable();
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                       SYSCTL_XTAL_16MHZ);
    //
    //使能用于开发板上 LED 的 GPIO 端口
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
```

```
//
//使能用于 LED 的 GPIO 引脚(PF2 和 PF3)
//
ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,GPIO_PIN_3 | GPIO_PIN_2);
//
//打开 UART0 并在 UART 上显示应用程序的名称
//
ConfigureUART();
UARTprintf("\033[2JTiva C Series USB bulk device example\n");
UARTprintf("-----------------------------\n\n");
//
//无需初始配置
//
g_bUSBConfigured = false;
//
//使能用于 USB 的 GPIO 外设并配置 USB 引脚
//
ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
ROM_GPIOPinTypeUSBAnalog(GPIO_PORTD_BASE,GPIO_PIN_4 | GPIO_PIN_5);
//
//使能系统时钟
//
ROM_SysTickPeriodSet(ROM_SysCtlClockGet()/SYSTICKS_PER_SECOND);
ROM_SysTickIntEnable();
ROM_SysTickEnable();
//
//打印 USB 配置
//
UARTprintf("Configuring USB\n");
//
//初始化发送和接收缓冲区
//
USBBufferInit(&g_sTxBuffer);
USBBufferInit(&g_sRxBuffer);
//
//设置带 VBUS 监视的 USB 设备的堆栈模式
//
USBStackModeSet(0,eUSBModeForceDevice,0);
//
//将设备信息传递到 USB 库,并把设备放置到总线上
//
USBDBulkInit(0,&g_sBulkDevice);
//
//等待初始配置完成
//
UARTprintf("Waiting for host...\n");
//
//清除计数器中的局部字节
```

```
//
ui32RxCount = 0;
ui32TxCount = 0;
//
//主要应用程序循环
//
while(1)
{
    //
    //查看是否有任何数据被传输
    //
    if((ui32TxCount != g_ui32TxCount) || (ui32RxCount != g_ui32RxCount))
    {
        //
        //自从上次检查之后,有任何传输吗?
        //
        if(ui32TxCount != g_ui32TxCount)
        {
            //
            //点亮绿色 LED
            //
            GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_3,GPIO_PIN_3);
            //
            //延迟
            //
            for(ui32Loop = 0; ui32Loop < 150000; ui32Loop ++ )
            {
            }
            //
            //熄灭绿色 LED
            //
            GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_3,0);
            //
            //Take a snapshot of the latest transmit count.
            //最新的发送计数快照
            //
            ui32TxCount = g_ui32TxCount;
        }
        //
        //自从上次检查之后,有任何接收流量吗?                    //
        if(ui32RxCount != g_ui32RxCount)
        {
            //
            //点亮蓝色 LED                          //
            GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2,GPIO_PIN_2);
            //
            //延迟
            //
            for(ui32Loop = 0; ui32Loop < 150000; ui32Loop ++ )
```

```
                    {
                    }
                    //
                    //熄灭蓝色 LED
                    //
                    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2,0);
                    //
                    //Take a snapshot of the latest receive count.
                    //最新的接收计数快照
                    //
                    ui32RxCount = g_ui32RxCount;
                }
                //
                //更新传输字节的显示
                //
                UARTprintf("\rTx: %d   Rx: %d",ui32TxCount,ui32RxCount);
            }
        }
    }
```

3. 加载 sub_dev_bulk 工程

在 C:\ti\TivaWare_C_Series－2.0.1.11577\examples\boards\EK－TM4C123GXL\usb_dev_bulk 目录下加载 sub_dev_bulk 工程。

4. 将工程 sub_dev_bulk 的编译结果加载到 LaunchPad 开发板中

将工程 sub_dev_bulk 的编译结果加载到 LaunchPad 开发板中，如图 14-19 所示。

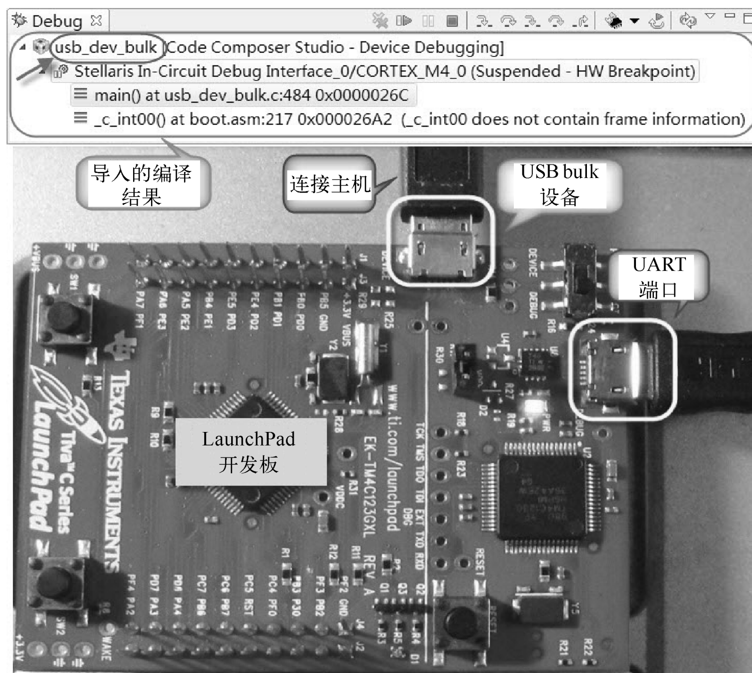


图 14-19 将编译结果导入到 LaunchPad 开发板中

5. 测试

1）打开 PuTTY（波特率为 115200 bit/s）。

2）发现新硬件（Generic Bulk Device）并添加其驱动程序（如图 14-20 所示）。



图 14-20　发现新硬件（Generic Bulk Device）并添加驱动程序过程

3）在 C:\Program Files\Texas Instruments\Stellaris\usb_examples 目录下启动 usb_bulk_example bulk 设备测试工具（如图 14-21 所示）。

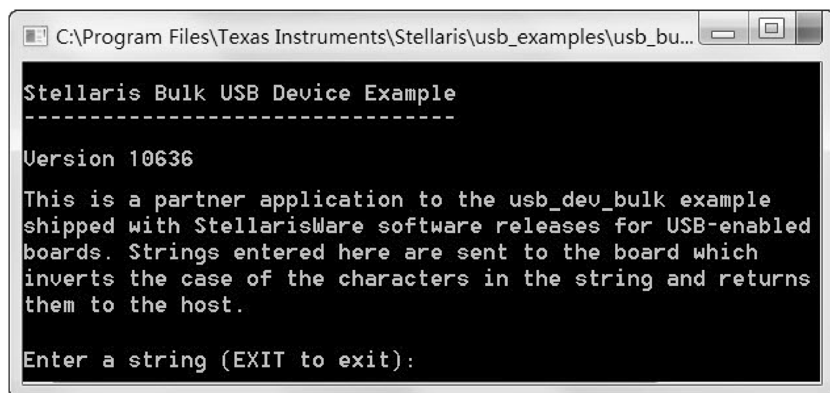


图 14-21　usb_bulk_example bulk 设备测试工具

4）单击工具栏中的图标，启动代码在 LaunchPad 开发板中运行，测试结果如图 14-22 所示。

从图 14-22 中可以看到，不但写入到缓冲区中的数据字节数和从缓冲区中读出的数据字节数相等，而且字母的大小写也发生了倒置。并且从上位机“usb_bulk_example”得出的测试结果和通过 UART 读出的测试结果一致，验证了上述程序实现了所要求的功能，程序设计正确。

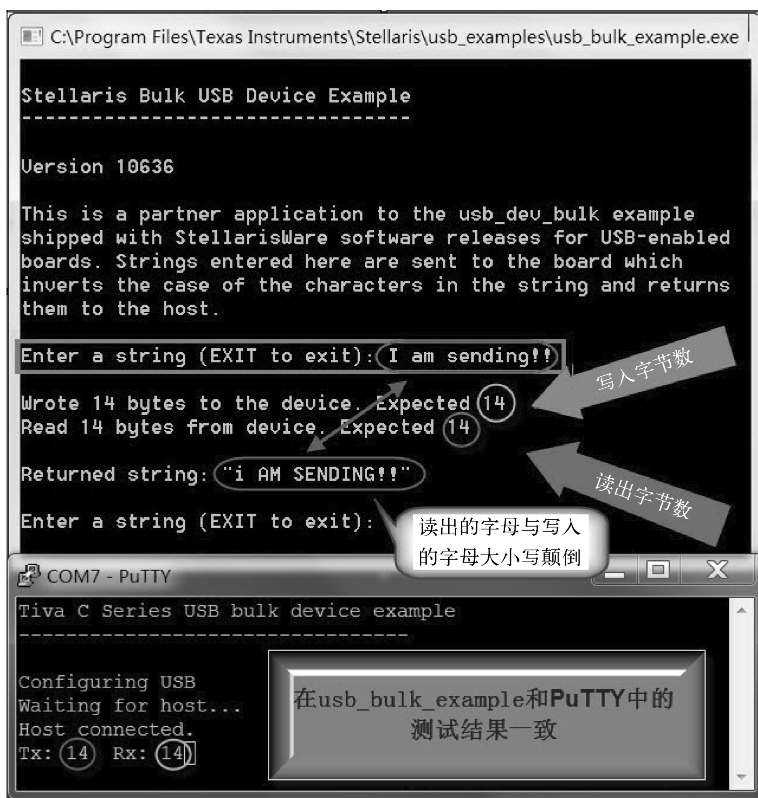


图 14-22　对 bulk 设备的测试结果

第 15 章

FatFS 文件读取实验

本章以 TI 提供的基于安全数码卡（Secure Digital Memory Card）的 FatFS 文件读取实验为例来介绍其使用方法，可使不熟悉 SD 卡（即安全数码卡）协议的读者能读懂 SD 卡的驱动程序。本章先简要介绍 SD 卡 2.0 的基本知识（这部分内容来自于 SD 卡 2.0 协议和网络），然后说明 TI 提供的 SD 卡驱动程序的编写方法，最后演示基于 SD 卡的 FatFS 文件系统的读取过程。

本章的主要内容：

- SD 卡介绍
- FatFS 文件系统简介
- 基于 SD 卡的文件读取实验



15.1　SD 卡概述

下面仅就比较常用的几种存储卡作一简单介绍：

MMC（MultiMedia Card）多媒体卡是一种非易失性存储器件，1997 年由西门子及 SanDisk 公司共同开发。SD 卡是在 MMC 的基础上发展而来的，由日本松下、东芝及 SanDisk 公司于 1999 年共同开发研制，它是基于半导体 Flash 的新一代记忆设备，仅一张邮票大小（如图 15-1 所示）。目前有 3 个版本的 SD 卡协议，SD 卡 2.0 版本根据容量的大小，可分为 SDSC 卡（即 SD 卡）（容量 < 2 GB）和 SDHC（2 GB < 容量 < 32 GB），且向前兼容 SD 卡 1.0 版和 MMC 卡，即所有支持 SD 卡的设备也支持 MMC 卡。

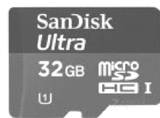


图 15-1　microSD 卡外形

15.1.1　SD 卡的内部结构及信号描述

1. SD 卡的内部结构

SD 卡的内部结构如图 15-2 所示。

2. SD 卡的寄存器描述

在图 15-2 中 SD 卡的左侧的寄存器描述见表 15-1。

3. SD 卡的引脚信号描述

SD 卡支持两工作模式，即 SD 模式（独立指令和数据通道，独有的传输格式）和 SPI 模式（独立序列输入和序列输出）。在基于 ARM Cortex 器件的应用中，一般采用 SPI 模式。SD 卡的引脚描述见表 15-2；实物图如图 15-3 所示。

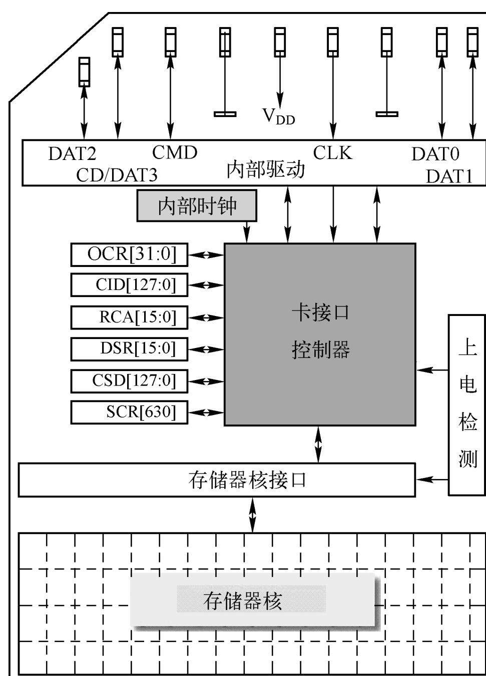


图 15-2　SD 卡的内部结构

表 15-1　SD 卡的寄存器描述

名称	宽度/bit	描　　述
CID	128	用于识别 SD 卡的卡识别号（强制）
RCA	16	SD 卡的相对地址（在 SPI 模式不可用）。卡的本地动态地址，可在卡的主机初始化过程中确认（强制）
DSR	16	驱动阶段寄存器，用于配置 SD 卡的输出驱动器（可选）
CSD	128	SD 卡特性数据，有关卡的操作条件信息（强制）
SDR	64	SD 卡配置寄存器，有关 microSD 存储卡的特殊功能的信息（强制）
OCR	32	操作条件寄存器（强制）

表 15-2　SD 卡的引脚描述

引脚	SD 模式			SPI 模式		
	名称	类型	描述	名称	类型	描述
1	CD/DAT3	I/O/PP	卡检测/数据线 3	CS	I	片选（低有效）
2	CMD	PP	命令/回复	DI	I	数据输入
3	Vss1	S	地	VSS	S	地
4	Vdd	S	供电电压	VDD	S	供电电压
5	CLK	I	时钟	CLK	I	时钟
6	Css2	S	地	VSS2	S	地
7	DAT0	I/O/PP	数据线 0	DO	O	数据输出
8	DAT1	I/O/PP	数据线 1	RSV	—	
9	DAT2	I/O/PP	数据线 2	RSV	—	

注：S 为电源供电，I 为输入，O 为输出，I/O/PP 为使用推挽驱动的输入输出。

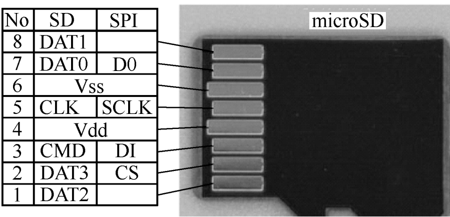


图 15-3　microSD 卡

15.1.2　SD 卡的命令

1. SD 卡的命令格式

SD 卡的命令格式见表 15-3。

表 15-3　SD 卡的命令格式

字节 1			字节 2～5（参数命令）	字节 6	
0	1	CMD	高位在前（参数可以为空）	CRC	1

例如 CMD0 命令格式为

0 1　000000　0000…00　1001010 1

固定　　命令　　　参数　　　CRC 码

　　　0x40　　0x00… 0x00　　0x95

2. SD 卡的命令

SD 卡的命令可共分为 12 类，即 Class0～Class11，其中部分命令集见表 15-4。

表 15-4　SD 卡命令分类及功能

命令	响应	缩　　写	功　　能
卡识别、初始化等基本命令集：Class0			
CMD0	R1	GO_IDLE_STATE	复位 SD 卡
CMD1	R1	SEND_OP_COND	激活卡的初始化过程（读 OCR 寄存器）
CMD8	R7	SEND_IF_COND	检查 SD 卡工作电压范围
CMD9	R1	SEND_CSD	读 CSD 寄存器
CMD10	R1	SEND_CID	读 CID 寄存器
CMD12	R1	STOP_TRANSMISSION	禁止读多块时的数据传输
CMD13	R2	SEND_STATUS	读卡状态寄存器
ACMD41	R1	SEND_OP_COND	发送主机容量支持信息，并激活该卡的初始化过程
CMD58	R3	READ_OCR	接到本指令后，卡将传送 OCR 数据
CMD59	R1	CRC_ON_OFF	设置 CRC 使能（1）或禁止（0）
读卡命令集：Class2			
命令	响应	缩　　写	功　　能
CMD16	R1	SET_BLOCK_LEN	设置块的长度
CMD17	R1	READ_SINGLE_BLOCK	读单块
CMD18	R1	READ_MULTIPLE_BLOCK	读多块，直到主机发送 CMD12 为止

（续）

写卡命令集：Class4			
命令	响应	缩　　写	功　　能
CMD24	R1	WRITE_BLOCK	写单块
CMD25	R1	WRITE_MULTIPLE_BLOCK	写多块
CMD27	R1	PROGRAM_CSD	写 CSD 寄存器
擦除卡命令集：Class5			
命令	响应	缩　　写	功　　能
CMD32	R1	ERASE_WR_BLK_START	设置块擦除的起始地址
CMD33	R1	ERASE_WR_BLK_END	设置块擦除的终止地址
CMD38	R1b	ERASE	选择要擦除的块
写保护命令集：Class6			
命令	响应	缩　　写	功　　能
CMD28	R1b	SET_WRITE_PROT	设置块写保护的地址
CMD29	R1b	CLR_WRITE_PROT	擦除块写保护的地址
CMD30	R1	SEND_WRITE_PROT	查询卡写保护位的状态
SD 卡锁定，锁解除功能命令集：Class7			
申请特定命令集：Class8			
命令	响应	缩　　写	功　　能
CMD55	R1	APP_CMD（所有应用命令之前必须先执行 CMD55）	告知卡的下一条命令是特殊命令，而非标准命令
CMD56	R1	GEN_CMD	应用相关（通用目的）的数据块读写命令
Class10、11：保留			

3. 命令响应格式

1）R1：每一个命令之后由卡发送该响应令牌（除 SEND_STATUS 命令外），其长度为 1 个字节，最高有效位（Most Significant Bit，MSB）总是设置为零，其他位为错误指示。R1 的响应格式如图 15-4 所示。

7	6	5	4	3	2	1	0
0	参数错误	地址错误	擦除顺序错误	CRC通信错误	非法命令	擦除复位	空闲状态

图 15-4　R1 响应格式

2）R2：由卡发送的响应 SEND_STATUS 命令的响应令牌（长度为 2 B），R2 的响应格式如图 15-5 所示。

7	6	5	4	3	2	1	0
0	擦除参考	写保护失当	卡ECC错误	CC错误	错误	跳过写保护擦除，锁定/解锁命令失败	卡锁定

7	6	5	4	3	2	1	0
0	参考错误	地址错误	擦除序列错误	通信CRC错误	非法命令	擦除复位	空闲状态

图 15-5　R2 响应格式

3）R3：当收到 READ_OCR 命令时，由卡发送的反应令牌，响应长度为 5 B。第 1

（MSB）字节的结构与响应类型 R1 相同，其他 4 B 包含 OCR 寄存器，如图 15-6 所示。

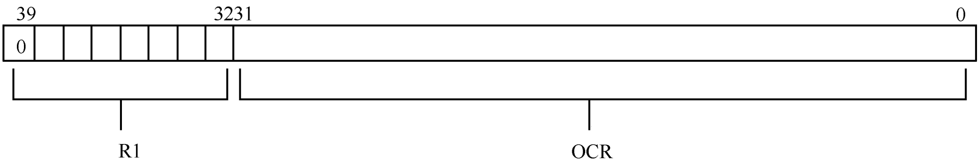


图 15-6　R3 的响应格式

15.1.3 SD 卡的功能描述

主机和卡之间的所有通信都是由主机控制的，主机所发送的命令包括广播和寻址（点对点）命令两种通信类型。

SD 卡的操作可分为两种模式：卡识别模式；数据传输模式。

表 15-5 表达了卡状态与操作模式之间的关系。

表 15-5　卡状态与操作模式

卡状态	操作模式
非激活状态	无动作
空闲状态	卡识别模式
就绪状态	
识别状态	
待机状态	数据传输模式
传输状态	
发送数据状态	
接收数据状态	
编程状态	
断开状态	

1. 卡识别模式

在该模式下，主机将检测 SD 卡的电压范围，识别 SD 卡类型，并要求其发送各自的相对地址，这一切操作都是通过命令线（CMD 线）来实现的，所有操作均采用默认的 SD 卡识别时钟频率。

（1）卡复位

当卡上电或收到 GO_IDLE_STATE（CMD0）命令之后，SD 卡都将进入空闲状态（非激活的卡除外）。即卡上电至少延迟 74 个时钟周期后方可进行总线传输（发送复位命令 CMD0），待复位完成后（即接收到 01h 响应，如图 15-12 所示），在得到响应 0x00 前需连续发送 CMD55 + ACMD41。当处于空闲状态时，SD 卡的 CMD 线处于输入模式，等待下一个命令的起始令牌。默认的相对地址 RCA 为 0x0000。

（2）工作条件验证

① 在主机和 SD 卡通信之前，主机并不知道 SD 卡支持的电压范围，这时主机先用一个特定的电压发送一条复位命令（CMD0），然后发送 SEND_IF_COND（CMD8）命令来获取 SD 卡所支持的电压范围。如果 SD 卡不支持所提供的电压，则不会发出任何响应信息，SD 卡将继续处于空闲状态。

② 在空闲（Idle）模式下，SD_SEND_OP_COND(ACMD41)命令旨在提供一种 SD 卡主机识别并拒绝与主机给定电压 VDD 不匹配卡的机制。主机通过发送命令操作数来作为所需电压 VDD 窗口的大小。若 SD 卡在指定范围内不能进行数据传输，应放弃进一步的总线操作，而进入非激活状态。注意：A 开头的命令可看作“Application Command”的伴随命令，在发任何一条 ACMD 命令之前，必须首先发一条 CMD55 命令，因此在发送 ACMD41 命令时，必须先行发送 CMD55 命令，然后方可发送 ACMD41 命令。

③ 在准备“就绪”模式下，主机将发送 ALL_SEND_CID(CMD2)命令到每个卡中来获取它们的唯一卡标识号（CID）。当卡发送完 CID 号后，SD 卡将进入识别状态。

④ 当主机发送 SEND_RELATIVE_ADDR(CMD3)命令来要求各个 SD 卡发送一个新的相对地址（RCA）时，RCA 用于其后在数据传输模式中的卡寻址。一旦获取 RCA，卡状态将变成待机状态。此时，如果主机想要给 SD 卡分配一个新的 RCA，只需发送另一条 CMD3 命令给 SD 卡即可，最后发布的 RCA 就是实际使用的 RCA 了。这个唯一地址发送给主机就进入了下一个阶段的数据传输模式，每张 SD 卡将与主机进行点对点的传输。

注意：当主机发出 CMD0 命令来复位 SD 卡时，应先发出 CMD8 命令再发送 ACMD41 命令来重新初始化 SD 卡。

SD 卡的状态图（卡识别模式）如图 15-7 所示；而 SD 卡的初始化与识别流程（SD 模式）如图 15-8 所示。

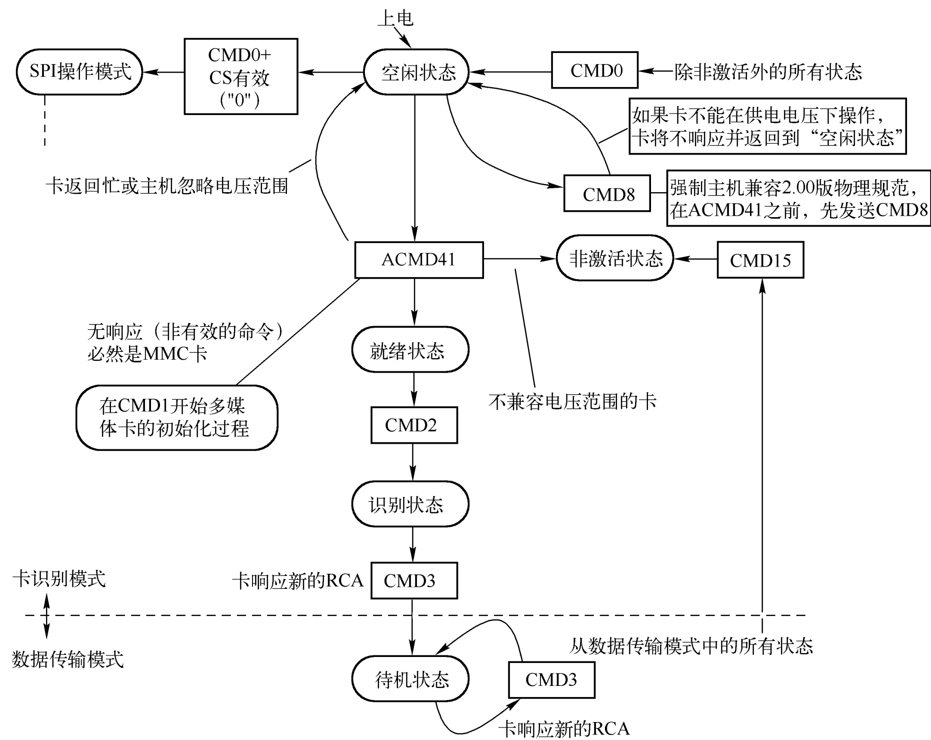


图 15-7　SD 卡的状态图（卡识别模式）

2. 数据传输模式

在 SD 卡识别模式结束之后，主机先不停地发送 SEND_CSD（CMD9）命令来获取卡的 CSD 信息，包括块长度、卡容量等。广播命令 SET_DSR(CMD4)为各个已识别的 SD 卡配置

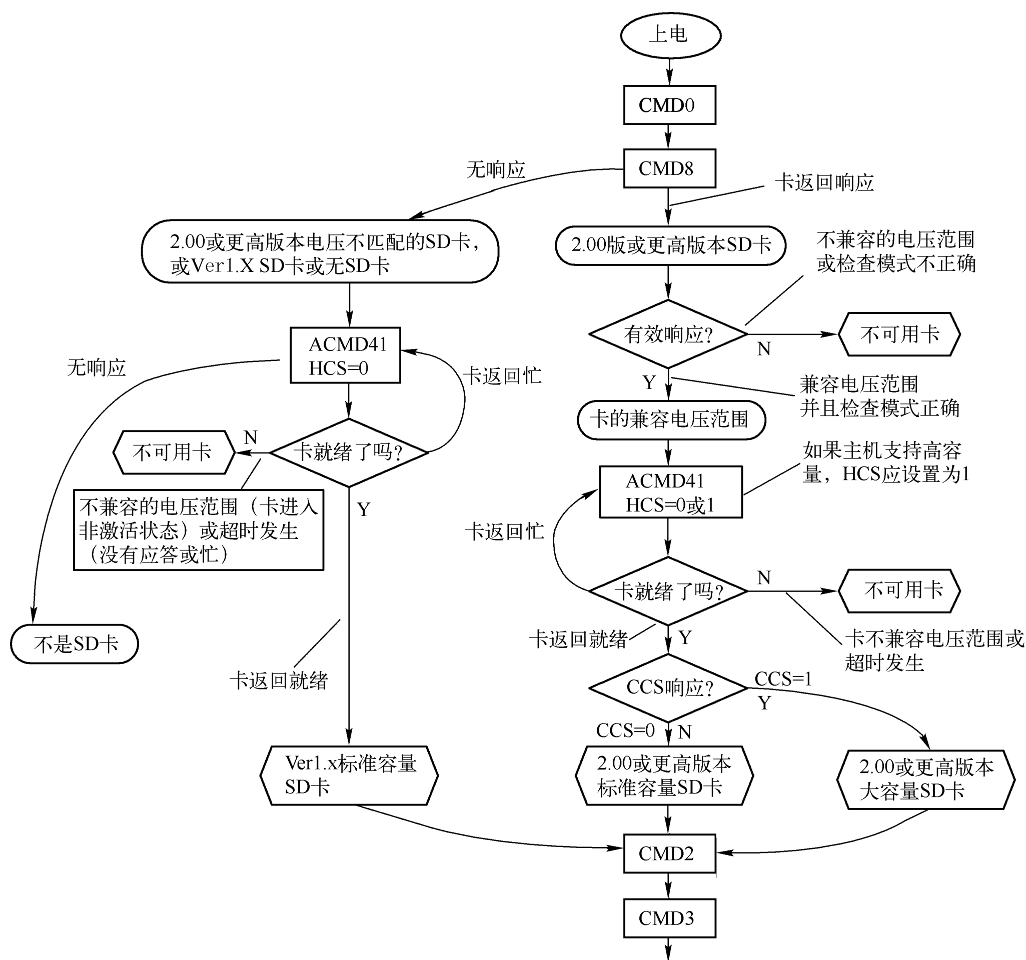


图 15-8　SD 卡的初始化与识别流程（SD 模式）

驱动阶段。它会向 SD 卡的 DSR 寄存器写入相关的信息，包括数据总线宽度、总线上卡的个数、总线频率等，这里 SET_DSR 命令是可选的。

CMD7 命令可使指定地址的 SD 卡进入传输模式，在指定时间段内，只有一个卡能处于传输状态。当某个原先被选中的处于传输状态的 SD 卡接收到 CMD7 命令后，会释放与主机的连接，并进入待机状态。当 CMD7 使用保留地址 0x0000 时，所有的 SD 卡都将进入待机状态。CMD13 用于得到卡的 CSD 的相关设置值；ACMD6 用于配置数据总线的宽度；读操作使用 CMD17、CMD18、CMD30、CMD56(r)、ACMD13、ACMD22、ACMD51 命令；写操作使用 CMD24、CMD25 CMD26、CMD27、CMD42、CMD56(w)命令；读/写停止操作使用 CMD12 命令。有关 SD 卡数据传输模式更多的信息如图 15-9 所示。

3. SD 卡的读写操作

(1) SD 卡的数据读取

在 SPI 模式下，读命令支持单块（CMD17）和多个块（CMD18）的读操作。由命令 CMD16 设定块的长度为 512 B。块数据读取操作如图 15-10 所示；读取单块数据流程如图 15-11 所示。

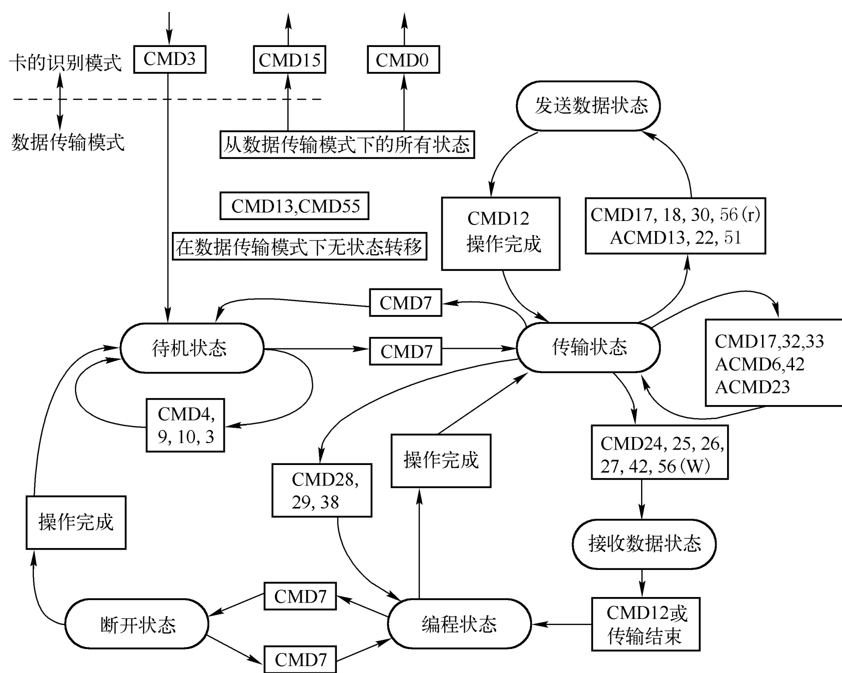


图 15-9　数据传输模式的 SD 卡状态图

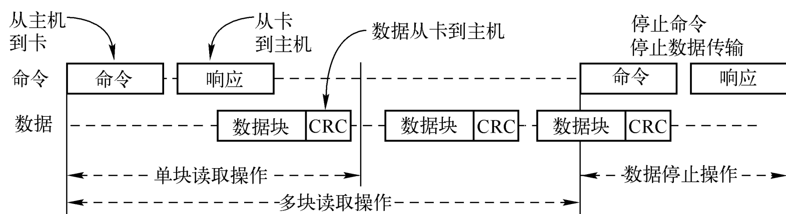


图 15-10　（多）块数据读取操作

（2）SD 卡的数据写入

在 SPI 模式下，写数据块命令将把单块（CMD24）或多块（CMD27）数据写入 SD 卡中，支持块写入操作的 SD 卡要求由命令 CMD16 来设置块的长度为 512 B。若出现 BLOCK_LEN_ERROR 或 ADDRESS_ERROR，写命令将被禁止。块数据写入操作如图 15-12 所示；写入单块数据的流程如图 15-13 所示。

注意：在进行 SD 卡的数据写操作时，最好首先对该卡进行擦除操作，然后再做写数据操作。

（3）SD 卡的数据擦除

在 SPI 模式下，首先发送 CMD32 命令和参数来指定要擦除的起始地址（即块号），然后再发送 CMD33 命令来指定结束地址，最后发送 CMD38 命令来擦除指定单元的数据，注意这 3 个操作步骤不可更改。

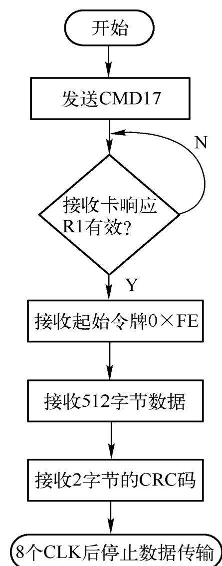


图 15-11　读取单块数据流程

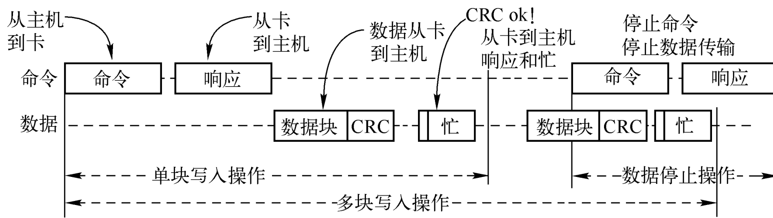


图 15-12 （多）块数据写入操作

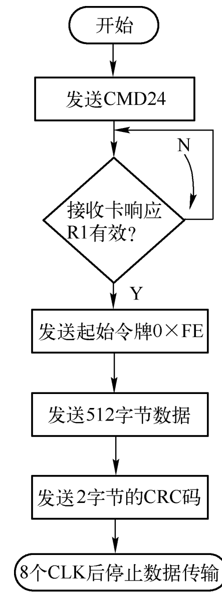


图 15-13 写入单块数据流程

15.1.4 SD 卡驱动程序解读

由于 SD 协议内容比较宽泛，如果对它没有较深刻的理解，想要写出 SD 卡的驱动程序也不是件容易的事。或许从芯片厂家提供的 SD 卡驱动程序入手，不失为一种多快好省的方法。下面将根据以上介绍的 SD 卡知识，对 TI 提供的 SD 卡驱动程序作一说明。

```
/* ------------------------------------------------------------------ */
/* 这段程序是对 FatFS 网站的样本进行修改得来的                          */
/* ------------------------------------------------------------------ */
#include <stdint.h>
…                                /* 包含文件略 */
#include "fatfs/src/diskio.h"
/* 定义 MMC/SDC 命令 */
#define CMD0        (0x40+0)     /* 复位 SD 卡。 */
#define CMD1        (0x40+1)     /* 读 OCR 寄存器 */
#define CMD8        (0x40+8)     /* SEND_IF_COND */
#define CMD9        (0x40+9)     /* 读 CSD 寄存器 */
#define CMD10       (0x40+10)    /* 读 CID 寄存器 */
#define CMD12       (0x40+12)    /* 禁止读多块时的数据传输 */
#define CMD16       (0x40+16)    /* 设置块的长度 */
#define CMD17       (0x40+17)    /* 读单块 */
#define CMD18       (0x40+18)    /* 读多块,直到主机发送 CMD12 为止 */
#define CMD23       (0x40+23)    /* 设置块数目 */
#define CMD24       (0x40+24)    /* 写单块 */
#define CMD25       (0x40+25)    /* 写多块 */
#define CMD41       (0x40+41)    /* 引用命令的前命令 */
#define CMD55       (0x40+55)    /* 所有应用命令之前必须先执行 CMD55 */
#define CMD58       (0x40+58)    /* 接到本指令后,卡将传送 OCR 数据 */
/*   DK-TM4C123G 板的 SSI 端口定义 */
```

```
#define SDC_SSI_BASE SSI0_BASE
#define SDC_SSI_SYSCTL_PERIPH SYSCTL_PERIPH_SSI0
//用于 SSI 的 GPIO 引脚
#define SDC_GPIO_PORT_BASE          GPIO_PORTA_BASE
#define SDC_GPIO_SYSCTL_PERIPH      SYSCTL_PERIPH_GPIOA
#define SDC_SSI_CLK                 GPIO_PIN_2
#define SDC_SSI_TX                  GPIO_PIN_5
#define SDC_SSI_RX                  GPIO_PIN_4
#define SDC_SSI_FSS                 GPIO_PIN_3
#define  SDC_SSI_PIN               (SDC_SSI_TX | SDC_SSI_RX | SDC_SSI_CLK | SDC_SSI_FSS)
//使卡的 CS 引脚有效,即 CS = 0 拉低
static
void SELECT (void)
{
    ROM_GPIOPinWrite(SDC_GPIO_PORT_BASE,SDC_SSI_FSS,0);
}
//使卡的 CS 引脚无效,即 CS = 1 拉高
static
void DESELECT (void)
{
    ROM_GPIOPinWrite(SDC_GPIO_PORT_BASE,SDC_SSI_FSS,SDC_SSI_FSS);
}
/* ------------------------------------------------------------------
  模块的私有函数
 ------------------------------------------------------------------ */
static volatile
DSTATUS Stat = STA_NOINIT;      /* 卡状态 */
static volatile
BYTE Timer1,Timer2;             /* 100 Hz 递减定时器 */
static
BYTE CardType;                  /* b0:MMC,b1:SDC,b2: 块寻址 */
static
BYTE PowerFlag = 0;             /* 电源接通指示 */
/* ------------------------------------------------------------------- */
/* 通过 SPI 发送一个字节到 MMC 卡(与平台相关)                            */
/* ------------------------------------------------------------------- */
static
void xmit_spi(BYTE dat)
{
    uint32_t ui32RcvDat;
    ROM_SSIDataPut(SDC_SSI_BASE,dat);              /* 向发送 FIFO 中写入数据 */
    ROM_SSIDataGet(SDC_SSI_BASE,&ui32RcvDat);      /* 在写入过程中刷新读取数据 */
}
/* ------------------------------------------------------------------- */
/* 通过 SPI 从 MMC 卡接收一个字节(与平台相关)                            */
/* ------------------------------------------------------------------- */
static
BYTE rcvr_spi (void)
{
```

```
    uint32_t ui32RcvDat;
    ROM_SSIDataPut(SDC_SSI_BASE,0xFF);              /*在写数据块前需写入若干空数据*/
    ROM_SSIDataGet(SDC_SSI_BASE,&ui32RcvDat);       /*从接收 FIFO 读取数据*/
    return (BYTE)ui32RcvDat;
}
static
void rcvr_spi_m (BYTE *dst)
{
    *dst = rcvr_spi();
}
/*-------------------------------------------------------------------*/
/*等待卡准备就绪                                                      */
/*-------------------------------------------------------------------*/
static
BYTE wait_ready (void)
{
    BYTE res;
    Timer2 =50;     /*等待准备就绪,500ms 超时*/
    rcvr_spi();
    do
        res = rcvr_spi();
    while ((res != 0xFF)&& Timer2);
    return res;
}
/*-------------------------------------------------------------------*/
/*发送 80 个左右的时钟脉冲给 CS 信号并使 DI(IN)为高电平。这是对卡上电后进入 SPI 模式的
要求,如图 15-14 所示*/
```

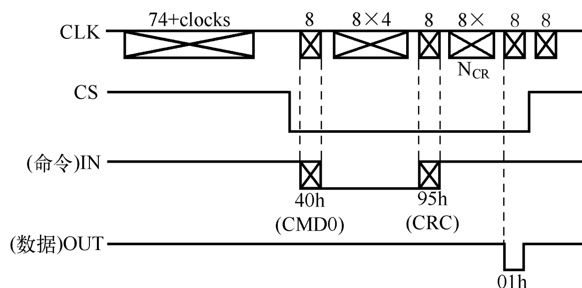


图 15-14　SD 卡复位的时序图

```
/*-------------------------------------------------------------------*/
static
void send_initial_clock_train(void)
{
    UINT i;
    uint32_t ui32Dat;
    /*确保 CS 保持高电平*/
    DESELECT();
    /*将 SSI TX 线切换到 GPIO 并使其为高电平*/
    ROM_GPIOPinTypeGPIOOutput(SDC_GPIO_PORT_BASE,SDC_SSI_TX);
```

```
    ROM_GPIOPinWrite(SDC_GPIO_PORT_BASE,SDC_SSI_TX,SDC_SSI_TX);
    /* SSI 发送 10 个字节空操作来使 SD 卡达到正常工作电压和进行同步 */
    for(i=0 ; i<10 ; i++)
    {
        /* 写空数据。直到 FIFO 中有多余空间 */
        ROM_SSIDataPut(SDC_SSI_BASE,0xFF);
        /* 数据写入过程中刷新读取数据 */
        ROM_SSIDataGet(SDC_SSI_BASE,&ui32Dat);
    }
    /* 返回 SSI TX 线的硬件控制 */
    ROM_GPIOPinTypeSSI(SDC_GPIO_PORT_BASE,SDC_SSI_TX);
}
/* ----------------------------------------------------------------- */
/* 电源控制(与平台相关)                                               */
/* ----------------------------------------------------------------- */
/* 当目标系统不支持电源插槽控制时,这些函数将无动作并且 chk_power 总是返回 1 */
static
void power_on (void)
{
    /* 这并不要真正打开电源,而只是初始化与卡会话的 SSI 端口和引脚 */
    /* 使能用于驱动 SDC 卡的 SSI 外设 */
    ROM_SysCtlPeripheralEnable(SDC_SSI_SYSCTL_PERIPH);
    ROM_SysCtlPeripheralEnable(SDC_GPIO_SYSCTL_PERIPH);
    /* 将 GPIO 引脚配置成 SSI 功能 */
    ROM_GPIOPinTypeSSI(SDC_GPIO_PORT_BASE,SDC_SSI_TX | SDC_SSI_RX | SDC_SSI_CLK);
    ROM_GPIOPinTypeGPIOOutput(SDC_GPIO_PORT_BASE,SDC_SSI_FSS);
    /* 设置 SSI 输出引脚为 4mA 推挽驱动并上拉到接收线上 */
    ROM_GPIOPadConfigSet(SDC_GPIO_PORT_BASE,SDC_SSI_RX,GPIO_STRENGTH_4MA,
                         GPIO_PIN_TYPE_STD_WPU);
    ROM_GPIOPadConfigSet(SDC_GPIO_PORT_BASE,SDC_SSI_CLK | SDC_SSI_TX | SDC_SSI_FSS,
                         GPIO_STRENGTH_4MA,GPIO_PIN_TYPE_STD);
    /* 配置 SSI0 端口 */
    ROM_SSIConfigSetExpClk(SDC_SSI_BASE,ROM_SysCtlClockGet(),
                          SSI_FRF_MOTO_MODE_0,SSI_MODE_MASTER,400000,8);
    ROM_SSIEnable(SDC_SSI_BASE);
    /* 设置 DI 和 CS 为高,为使 SD 卡正常工作至少延迟 74 个 SCLK 脉冲(如图 15-14 所示) */
    /* 接收本地命令 */
    send_initial_clock_train();
    PowerFlag=1;
}
//设置 SSI 速度到最大
static
void set_max_speed(void)
{
    uint32_t i;
    /* 关闭 SSI  */
    ROM_SSIDisable(SDC_SSI_BASE);
    /* 设置最大时钟频率为系统时钟的一半,即 12.5 MHz */
```

```
    i = ROM_SysCtlClockGet( )/2;
    if( i > 12500000)
    {
        i = 12500000;
    }
    /* 配置 SSI0 端口以 12.5 MHz 运行 */
    ROM_SSIConfigSetExpClk( SDC_SSI_BASE, ROM_SysCtlClockGet( ),
                                        SSI_FRF_MOTO_MODE_0, SSI_MODE_MASTER, i, 8);
    /* 使能 SSI  */
    ROM_SSIEnable( SDC_SSI_BASE);
}
static
void power_off (void)
{
    PowerFlag = 0;
}
static
int chk_power( void)              /* 插槽的电源状态:0 = 关,1 = 开 */
{
    return PowerFlag;
}
/* ------------------------------------------------------------------------ */
/* 从 MMC 接收数据包,如图 15-15 所示(包括参考图 15-10、图 15-11)
```

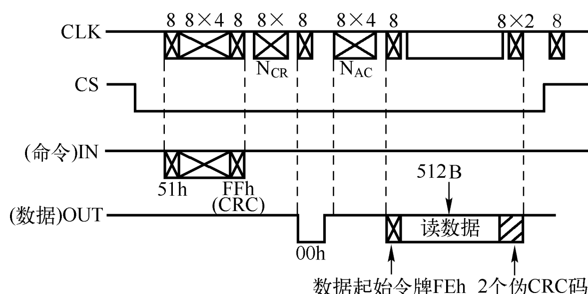


图 15-15　SD 卡读操作时序

```
/* ------------------------------------------------------------------------ */
static
BOOL rcvr_datablock (
    BYTE *buff,                               /* 保存接收数据的数据缓冲区 */
    UINT btr                                  /* 字节数(必须是偶数) */
)
{
    BYTE token;
    Timer1 = 100;
    do {                                      /* 等待数据包的 100ms 超时(SD 卡协议要求) */
        token = rcvr_spi( );
    } while ((token == 0xFF) && Timer1);      /* 如图 15-15 中 SD 卡读操作时序 */
    if(token != 0xFE) return FALSE;           /* 如果不是数据起始令牌,将返回错误 */
    do {                                      /* 将收到的数据块存入缓冲区中 */
        rcvr_spi_m( buff ++ );
```

```
        rcvr_spi_m( buff ++ ) ;
    } while ( btr - =2) ;
    rcvr_spi( ) ;                       /* 丢弃读取的两个字节的 CRC 码 */
    rcvr_spi( ) ;
    return TRUE;                        /* 返回成功 */
}
/* ------------------------------------------------------------------------- */
/* 发送一个数据包到 MMC 卡中,如图 15-16 所示(包括图 15-12、图 15-13)
```

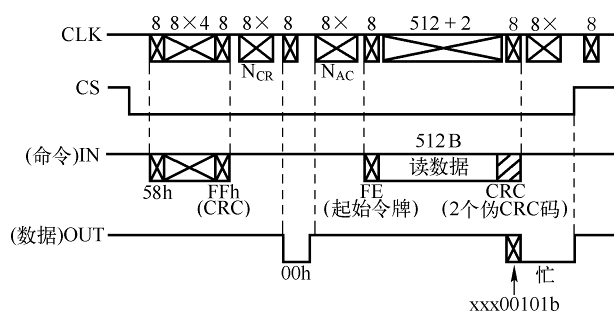


图 15-16 SD 写入操作时序

```
/* ------------------------------------------------------------------------- */
#if _READONLY ==0
static
BOOL xmit_datablock (
    const BYTE * buff,              /* 要发送 512 字节的数据块,见图 15-16 */
    BYTE token                      /* 数据/停止令牌 */
)
{
    BYTE resp,wc;
    if (wait_ready( ) ! =0xFF)return FALSE;
    xmit_spi(token) ;               /* 发送数据令牌 */
    if (token ! =0xFD) {            /* 数据令牌 */
        wc =0;
        do {                        /* 发送 512 字节的数据块到 MMC */
            xmit_spi( * buff ++ ) ;
            xmit_spi( * buff ++ ) ;
        } while ( --wc) ;
        xmit_spi(0xFF) ;            /* 发送 2 个伪 CRC 码 */
        xmit_spi(0xFF) ;
        resp = rcvr_spi( ) ;        /* 接收数据响应 */
        if ((resp & 0x1F) ! =0x05)  /* 如果写入 512 字节未被接收,
            return FALSE;           则返回错误,见图 15-16 中的 SD 卡数据输出线 */
    }
    return TRUE;
}
#endif                              /* 只读(_READONLY) */
/* ------------------------------------------------------------------------- */
/* 发送命令包到 MMC 卡,见表 15-6
```

表 15-6　SD 卡的命令格式

字节 1			字节 2 ~5(参数命令)	字节 6	
0	1	CMD	高位在前(参考可以为空)	CRC	1

```
*/
/* ----------------------------------------------------------------- */
static
BYTE send_cmd (
    BYTE cmd,                          /* 命令字节 */
    DWORD arg                          /* 参数 */
)
{
    BYTE n,res;
    if (wait_ready()! =0xFF)return 0xFF;
    /* 发送命令包(CDM +4 个字节的参数 +2 个伪 CRC 码,见表 15-3) */
    xmit_spi(cmd);                     /* 命令 */
    xmit_spi((BYTE)(arg >> 24));       /* 参数[31 ~24] */
    xmit_spi((BYTE)(arg >> 16));       /* 参数[23 ~16] */
    xmit_spi((BYTE)(arg >> 8));        /* 参数[15 ~8] */
    xmit_spi((BYTE)arg);               /* 参数[7 ~0] */
    n =0xff;
    if (cmd == CMD0)n =0x95;           /* CMD0(0,即参数为 0)的 CRC 码 */
    if (cmd == CMD8)n =0x87;           /* CMD8(0x1AA,即参数为 0x1AA)的 CRC 码 */
    xmit_spi(n);
    /* 接收命令响应 */
    if (cmd == CMD12)rcvr_spi();       /* 当停止读数时跳过一个填充字节 */
    n =10;                             /* 等待有效响应在 10 次尝试后超时 */
    do
        res = rcvr_spi();
    while ((res & 0x80)&& --n);
    return res;                        /* 返回响应值 */
}
/* ----------------------------------------------------------------- *
* 发送用于终止多扇区读取的特殊命令,见表 15-7
```

表 15-7　SD 卡的命令格式

字节 1			字节 2 ~5(参数命令)	字节 6	
0	1	CMD	高位在前(参考可以为空)	CRC	1

```
* ---------------------------------------------------------------- */
static
BYTE send_cmd12 (void)
{
    BYTE n,res,val;
    /* 对于 CMD12,在发送新的命令前不必等待该卡进入空闲状态 */
    /* 发送命令包 - CMD12 的参数将被忽略 */
    xmit_spi(CMD12);
    xmit_spi(0);
    xmit_spi(0);
```

```
    xmit_spi(0);
    xmit_spi(0);
    xmit_spi(0);
    /*从卡中最多读取10个字节的数据,记住哪些读出的值不是0xFF的数据*/
    for(n=0; n<10; n++)
    {
        val=rcvr_spi();
        if(val != 0xFF)
        {
            res=val;
        }
    }
    return res;                               /*返回响应值*/
}
/*-----------------------------------------------------------------
公共函数
-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
/*卡的初始化                                                        */
/*-----------------------------------------------------------------*/
                */
DSTATUS disk_initialize (
    BYTE drv                                  /*物理驱动器号(0)*/
)
{
    BYTE n,ty,ocr[4];
    if (drv)return STA_NOINIT;                /*只支持单个驱动器*/
    if (Stat & STA_NODISK)return Stat;        /*卡没有插入*/
    power_on();                               /*接通卡槽的电源*/
    send_initial_clock_train();               /*确保卡处于SPI模式(延迟80个时钟)*/
    SELECT();                                 /*将片选(CS)信号拉低 */
    ty=0;
    if (send_cmd(CMD0,0)==1){                 /*进入空闲状态(如图15-7、图15-8所示)*/
        Timer1=100;                           /*100 ms初始化超时*/
        if (send_cmd(CMD8,0x1AA)==1){          /*SDC Ver2+,SD卡强制规范,如图15-7、
                                                图15-8中CMD8的说明,其中0x1AA为4字节
                                                的参数 */
            for (n=0; n<4; n++)ocr[n]=rcvr_spi();       /*读取OCR寄存器的值*/
            if (ocr[2]==0x01 && ocr[3]==0xAA){ /*卡的VDD范围为2.7~3.6V,查阅
                                                  CMD8寄存器描述 */
                do {
                    if (send_cmd(CMD55,0)<=1 && send_cmd(CMD41,\
                        1UL<<30)==0)break;    /*设置ACMD41的HCS位(30位)(即区
                                                别卡的类型,1=SDHC卡,0=SDSC卡)*/
                } while (Timer1);
                if (Timer1 && send_cmd(CMD58,0)==0){    /*SD卡2.0初始化成功,检查
                                                          CCS位,如图15-8所示*/
                    for (n=0; n<4; n++)ocr[n]=rcvr_spi();
                    ty=(ocr[0] & 0x40)?6:2;
```

```
                }
            }
        } else {                          /* 无响应,则 SD 卡为 SDC 1.x 版本 或 MMC 的卡 */
            ty = (send_cmd(CMD55,0) <= 1 && send_cmd(CMD41,0) <= 1) ? 2 : 1;
                                          /* SDC: MMC 卡(如图 15-8 所示) */
            do {
                if (ty == 2) {
                    if (send_cmd(CMD55,0) <= 1 && send_cmd(CMD41,0) == 0) break;
                                          /* ACMD41(如图 15-8 所示) */
                } else {
                    if(send_cmd(CMD1,0) == 0) break;     /* CMD1,另外初始化 MMC 卡 */
                }
            } while (Timer1);
            if (!Timer1 || send_cmd(CMD16,512) != 0)     /* 选择 R/W 模块长度 */
                ty = 0;
        }
    }
    CardType = ty;
    DESELECT();                           /* 拉高 CS 信号,即 CS = H */
    rcvr_spi();                           /* 空闲(释放 DO) */
    if (ty) {                             /* 初始化成功 */
        Stat &= ~STA_NOINIT;              /* 清除 STA_NOINIT */
        set_max_speed();                  /* 时钟频率应小于 400 kHz */
    } else {                              /* 初始化失败 */
        power_off();
    }
    return Stat;
}
/* --------------------------------------------------------------------- */
/* 获取 SD 卡状态                                                          */
/* --------------------------------------------------------------------- */
DSTATUS disk_status (
    BYTE drv                                      /* 物理驱动器号(0) */
)
{
    if (drv) return STA_NOINIT;                   /* 只支持单个驱动器 */
    return Stat;
}
/* --------------------------------------------------------------------- */
/* 读扇区(见图 15-9 ~ 11)
/* --------------------------------------------------------------------- */
DRESULT disk_read (
    BYTE drv,                                     /* 物理驱动器号(0) */
    BYTE *buff,                                   /* 指向存储读出数据的缓冲区指针 */
    DWORD sector,                                 /* 起始扇区号(LBA) */
    BYTE count                                    /* 扇区计数(1 ~ 255) */
)
{
    if (drv || !count) return RES_PARERR;
```

```
    if (Stat & STA_NOINIT) return RES_NOTRDY;
    if (!(CardType & 4)) sector * =512;           /* 如果需要可转换为字节地址 */
    SELECT();                                     /* CS = L */
    if (count == 1) {                             /* 读单块 */
        if ((send_cmd(CMD17, sector) == 0)        /* 读单个块(如图 15-9 所示) */
            && rcvr_datablock(buff,512))
            count = 0;
    }
    else {                                        /* 多个块 */
        if (send_cmd(CMD18, sector) == 0) {       /* 读多个块 */
            do {
                if (!rcvr_datablock(buff,512)) break;
                buff += 512;
            } while (--count);
            send_cmd12();                         /* 停止传输 */
        }
    }
    DESELECT();                                   /* CS = H */
    rcvr_spi();                                   /* 空闲 (释放 DO) */
    return count ? RES_ERROR: RES_OK;
}
/* -------------------------------------------------------------------- */
/* 写扇区 (如图 15-9、图 15-12、图 15-13 所示)                              */
/* -------------------------------------------------------------------- */
#if _READONLY == 0
DRESULT disk_write (
    BYTE drv,                                     /* 物理驱动器号(0) */
    const BYTE *buff,                             /* 指向要写入数据的指针 */
    DWORD sector,                                 /* 起始扇区号(LBA) */
    BYTE count                                    /* 扇区计数(1 ~255) */
)
{
    if (drv || !count) return RES_PARERR;
    if (Stat & STA_NOINIT) return RES_NOTRDY;
    if (Stat & STA_PROTECT) return RES_WRPRT;
    if (!(CardType & 4)) sector * =512;           /* 如果需要可转换为字节地址 */
    SELECT();                                     /* CS = L */
    if (count == 1) {                             /* 单块写入(如图 15-12 所示) */
        if ((send_cmd(CMD24, sector) == 0)        /* 单块写入 */
            && xmit_datablock(buff,0xFE))
            count = 0;
    }
    else {                                        /* 多块写 */
        if (CardType & 2) {
            send_cmd(CMD55,0); send_cmd(CMD23, count);    /* ACMD23 */
        }
        if (send_cmd(CMD25, sector) == 0) {               /* 多块写 */
            do {
                if (!xmit_datablock(buff,0xFC)) break;
```

```
                buff += 512;
            } while ( -- count);
            if ( !xmit_datablock(0,0xFD))                        /*停止传输令牌 */
                count = 1;
        }
    }
    DESELECT();                            /* CS = H */
    rcvr_spi();                            /*空闲(释放 DO)*/
    return count ? RES_ERROR: RES_OK;
}
#endif/*只读*/
/* -------------------------------------------------------------------------------- */
/*多种功能                                                                          */
/* -------------------------------------------------------------------------------- */
DRESULT disk_ioctl (
    BYTE drv,                           /*物理驱动器号(0)*/
    BYTE ctrl,                          /*控制代码*/
    void *buff                          /*发送/接收控制数据缓冲区*/
)
{
    DRESULT res;
    BYTE n,csd[16], *ptr = buff;
    WORD csize;
    if (drv)return RES_PARERR;
    res = RES_ERROR;
    if (ctrl == CTRL_POWER){
        switch ( *ptr){
        case 0:                         /*子控制代码 ==0 (电源关断)*/
            if (chk_power())
                power_off();            /*关断电源 */
            res = RES_OK;
            break;
        case 1:                         /*子控制代码 ==1 (电源接通)*/
            power_on();                 /* Power on */
            res = RES_OK;
            break;
        case 2:                         /*子控制代码 ==2 (获得电源)*/
            *(ptr+1) = (BYTE)chk_power();
            res = RES_OK;
            break;
        default:
            res = RES_PARERR;
        }
    }
    else {
        if (Stat & STA_NOINIT)return RES_NOTRDY;
        SELECT();                       /* CS = L */
        switch (ctrl){
        case GET_SECTOR_COUNT:                  /*获取卡的上扇区数(DWORD)*/
```

```
            if ((send_cmd(CMD9,0) ==0)&& rcvr_datablock(csd,16)){
                if ((csd[0] >>6) ==1){           /* SDC ver 2.00  */
                    csize = csd[9] + ((WORD)csd[8] <<8) +1;
                    *(DWORD *)buff = (DWORD)csize << 10;
                } else {                         /* MMC 或 SDC ver 1.xx  */
                    n = (csd[5] & 15) + ((csd[10] & 128) >>7) +    \
                        ((csd[9] & 3) <<1) +2;
                    csize = (csd[8] >>6) + ((WORD)csd[7] <<2) + \
                            ((WORD)(csd[6] & 3) <<10) +1;
                    *(DWORD *)buff = (DWORD)csize << (n -9);
                }
                res = RES_OK;
            }
            break;
            case GET_SECTOR_SIZE:                /* 得到扇区(WORD) */
            *(WORD *)buff =512;
            res = RES_OK;
            break;
        case CTRL_SYNC:                          /* 确保数据已写入 */
            if (wait_ready() ==0xFF)
                res = RES_OK;
            break;
        case MMC_GET_CSD:                        /* 接收 CSD 作为一个数据块(16 字节) */
            if (send_cmd(CMD9,0) ==0             /* 读 CSD  */
                && rcvr_datablock(ptr,16))
                res = RES_OK;
            break;
        case MMC_GET_CID:                        /* 接收 CID 作为一个数据块(16 字节) */
            if (send_cmd(CMD10,0) ==0            /* 读 CID  */
                && rcvr_datablock(ptr,16))
                res = RES_OK;
            break;
        case MMC_GET_OCR:                        /* 接收 OCR 作为 R3 的响应(4 字节) */
            if (send_cmd(CMD58,0) ==0){          /* 读 OCR  */
                for (n =0; n <4; n ++)
                    * ptr ++= rcvr_spi();
                res = RES_OK;
            }
//          case MMC_GET_TYPE:                   /* 获取卡的类型标志(1 字节) */
//              * ptr = CardType;
//              res = RES_OK;
//              break;
        default:
            res = RES_PARERR;
        }
        DESELECT();                              /* CS = H  */
        rcvr_spi();                              /* 空闲(释放 DO) */
    }
    return res;
```

```
}

/* --------------------------------------------------------------------------------- */
/* 设备定时器中断程序(依赖于平台)                                                   */
/* --------------------------------------------------------------------------------- */
/* 这个函数必须在 10 ms 之内调用                                                     */

void disk_timerproc (void)
{
//      BYTE n,s;
    BYTE n;
    n = Timer1;                         /* 100Hz 递增定时器 */
    if (n)Timer1 = --n;
    n = Timer2;
    if (n)Timer2 = --n;
}
/* --------------------------------------------------------------------------------- */
/* 用户给 FatFS 模块提供的定时函数                                                   */
/* --------------------------------------------------------------------------------- */
/* 这是一个从 FatFS 模块调用的实时时钟服务 */
/* 任何有效时间必须被返回,即使系统不支持实时时钟 */
DWORD get_fattime (void)
{
    return    ((2007UL - 1980) << 25)      //年 = 2007
              | (6UL << 21)                //月 = June
              | (5UL << 16)                //日 = 5
              | (11U << 11)                //时 = 11
              | (38U << 5)                 //分 = 38
              | (0U >> 1)                  //秒 = 0
              ;
}
```



15.2　SD 卡 FatFS 文件读取实验

15.2.1　FatFS 文件系统简介

FatFS 是小型嵌入式系统的通用 FAT 文件系统模块。FatFS 由符合 ANSI C 的 C 语言编写并和磁盘 I/O 层完全分离，因此它是独立于硬件架构的，如图 15-17 所示。

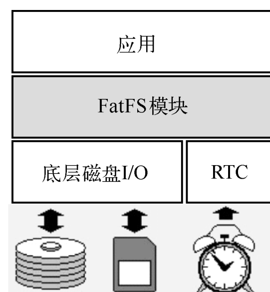


图 15-17　文件系统框图

1. 特点

1）Windows 兼容的 FAT 文件系统。

2）独立于平台易于移植。

3）代码和工作区占用空间很小。

4）各种配置选项。

5）多卷（物理驱动器和分区）。

① 多 ANSI/OEM 代码页，包括 DBCS。
② 支持在 ANSI/OEM 或 Unicode 中的长文件名。
③ RTOS 支持。
④ 多扇区大小支持。
⑤ 只读，最少 APII/O 与缓冲区等。

2. 应用程序接口

1）FatFS 文件系统的源文件夹中的文件如下：
① cc936c：为支持简体中文的程序，其包括简体中文的 GBK 和转换函数。
② diskio. h：底层磁盘接口模块中的包含文件。
③ diskio. c：FatFS 文件系统中，底层磁盘 I/O 模块骨架的实现代码。需用户根据实际情况自行修改。
④ integer. h：FatFS 模块的整数类型定义。
⑤ ff. h：R0. 09bFAT 文件系统模块的包含文件，需用户按实际情况稍作修改。
⑥ ff. c：R0. 09bFAT 文件系统模块的实现代码。
⑦ ffconf. h：FAT 文件系统的模块配置文件，需用户自行修改。
2）实现 FatFS 模块的固件库函数见表 15-8。

表 15-8　FatFS 固件库函数列表

函　　数	描　　述	函　　数	描　　述
f_open	打开/创建一个文件	f_chdir	改变当前目录
f_close	关闭打开的文件	f_chdrive	改变当前驱动器
f_read	读取文件	f_getcwd	检索当前目录
f_write	写入文件	f_getfree	获得空闲簇的个数
f_lseek	移动读/写指针来扩展文件大小	f_getlabel	获取卷标
f_truncate	截断文件大小	f_setlabel	设置卷标
f_sync	刷新缓存的数据	f_mount	注册/注销一个工作区
f_forward	读取文件数据并转发到数据流	f_mkfs	在磁盘上创建一个文件系统
f_stat	检查存在的文件或子目录	f_fdisk	划分物理驱动器
f_opendir	打开一个目录	f_gets	读一个字符串
f_closedir	关闭打开的目录	f_putc	写一个字符
f_readdir	读取目录项	f_puts	写一个字符串
f_mkdir	创建一个子目录	f_printf	写一个格式化字符串
f_unlink	删除一个文件或子目录	f_tell	获取当前的读/写指针
f_chmod	更改属性	f_eof	测试一个文件结束
f_utime	更改时间戳	f_size	获取文件的大小
f_rename	重命名/移动文件或子目录	f_error	测试一个文件中的错误

注：FatFS 文件系统网址为 http://elm - chan. org/fsw/ff/00index_e. html。

15. 2. 2　实验硬件连接图

图 15-18 是 TI 公司 BD - LM4F232 评估板的 microSD 卡的硬件连接图（对于仅有 LaunchPad：EK - TM4C123GXL 板的读者，可以参照该连线图在面包板中给 LaunchPad 板扩展一个 microSD 卡槽），采用 SSI 模式实现与微控制器通信。然后创建一个能显示 SD 卡访问操作的 UART 控制台，即通过上位机发送命令来访问 microSD 卡。

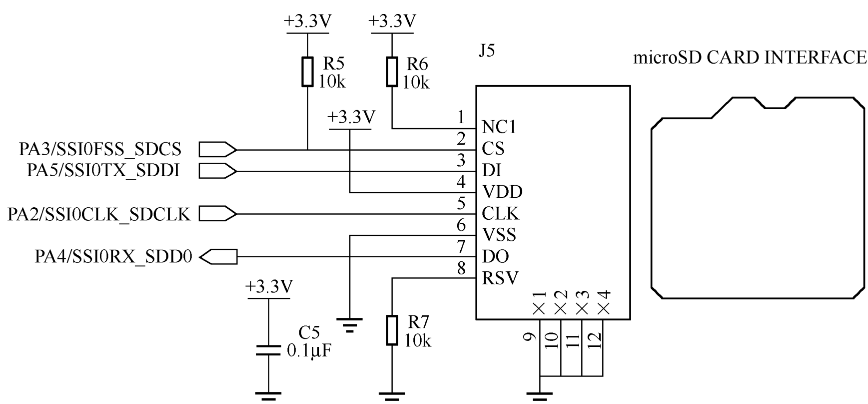


图 15-18　SD 卡的硬件连接图

15.2.3　导入 sd_card 工程

1）FatFS 文件读取程序说明。下面将以 TI 提供的一个简单的 FatFS 读取为例来介绍程序的编写。

```
//*****************************************************************************
//文件名:sd_card.c
//来源:TI 例程
//功能描述:此例程演示从 SD 卡上读取文件系统。采用 FAT32 文件系统驱动的 FatFS,可通过 UART
//串口命令来查看操作 SD 卡上的文件系统,包括在 PuTTY 或串口助手上输入 help 命令获取帮助信息
//*****************************************************************************
//…                                        //包含文件略
#include "utils/cmdline.h"
#include "utils/uartstdio.h"
#include "fatfs/src/ff.h"
#include "fatfs/src/diskio.h"
//*****************************************************************************
//定义了保存路径的缓冲区大小,或来自于 SD 卡的临时数据大小
//分配这两个缓冲区的大小必须足够大,以便能保存所需的完整路径名
//包括文件名和结尾的空字符
//*****************************************************************************
#define PATH_BUF_SIZE              80
//*****************************************************************************
//定义缓冲区大小来保存命令行
//*****************************************************************************
#define CMD_BUF_SIZE               64
//*****************************************************************************
//这个缓冲区保存当前工作目录的完整路径,其中"/"为根目录
//*****************************************************************************
static char g_pcCwdBuf[PATH_BUF_SIZE] = "/";
//*****************************************************************************
//在操作文件路径时会使用到临时数据缓冲区,或从 SD 卡中读取数据
//*****************************************************************************
static char g_pcTmpBuf[PATH_BUF_SIZE];
```

```
// ************************************************************************
//保存命令行的缓冲区
// ************************************************************************
static char g_pcCmdBuf[CMD_BUF_SIZE];
// ************************************************************************
//下面是所用 FatFS 的数据结构
// ************************************************************************
static FatFS        g_sFatFs;
static DIR          g_sDirObject;
static FILINFO      g_sFileInfo;
static FIL          g_sFileObject;
// ************************************************************************
//保存 fresult 数值代码之间的映射结构,以及字符串表示形式
//从 FatFS FAT 文件系统驱动程序中返回 FRESULT 代码
// ************************************************************************
typedef struct
{
    FRESULT iFResult;
    char  * pcResultStr;
}
tFResultString;
// ************************************************************************
//定义一个宏可很容易将结果代码添加到表中
// ************************************************************************
#define FRESULT_ENTRY(f)            { (f),(#f)}
// ************************************************************************
//一个保存数值 FRESULT 代码之间映射的表与其名称的字符串
//这可用于在 UART 控制台上打印查找到的错误代码
// ************************************************************************
tFResultString g_psFResultStrings[ ] =
{
    FRESULT_ENTRY(FR_OK),
    FRESULT_ENTRY(FR_DISK_ERR),
    FRESULT_ENTRY(FR_INT_ERR),
    FRESULT_ENTRY(FR_NOT_READY),
    FRESULT_ENTRY(FR_NO_FILE),
    FRESULT_ENTRY(FR_NO_PATH),
    FRESULT_ENTRY(FR_INVALID_NAME),
    FRESULT_ENTRY(FR_DENIED),
    FRESULT_ENTRY(FR_EXIST),
    FRESULT_ENTRY(FR_INVALID_OBJECT),
    FRESULT_ENTRY(FR_WRITE_PROTECTED),
    FRESULT_ENTRY(FR_INVALID_DRIVE),
    FRESULT_ENTRY(FR_NOT_ENABLED),
    FRESULT_ENTRY(FR_NO_FILESYSTEM),
    FRESULT_ENTRY(FR_MKFS_ABORTED),
    FRESULT_ENTRY(FR_TIMEOUT),
    FRESULT_ENTRY(FR_LOCKED),
    FRESULT_ENTRY(FR_NOT_ENOUGH_CORE),
```

```
    FRESULT_ENTRY(FR_TOO_MANY_OPEN_FILES),
    FRESULT_ENTRY(FR_INVALID_PARAMETER),
};
//****************************************************************************
//定义保存结果代码个数的宏
//****************************************************************************
#define NUM_FRESULT_CODES     (sizeof(g_psFResultStrings)/                    \
                               sizeof(tFResultString))
//****************************************************************************
//图形上下文用来在 CSTN 显示屏上显示文字
//****************************************************************************
tContext g_sContext;
//****************************************************************************
//在调用 FatFS 函数时该函数返回一个错误代码的字符串表示形式,可用于打印可读的错误消息
//****************************************************************************
const char *
StringFromFResult(FRESULT iFResult)
{
    uint_fast8_t ui8Idx;
    //进入一个循环来搜索与错误代码表中匹配的错误代码
    for(ui8Idx = 0; ui8Idx < NUM_FRESULT_CODES; ui8Idx ++ )
    {
        //如果找到一个匹配,则返回错误代码的字符串名称
        if(g_psFResultStrings[ui8Idx].iFResult == iFResult)
        {
            return(g_psFResultStrings[ui8Idx].pcResultStr);
        }
    }
    //若在该点上没有发现匹配的代码,则返回一个指示未知错误的字符串
    return("UNKNOWN ERROR CODE");
}
//****************************************************************************
//处理 SysTick 的中断。FatFS 因内部定时的目的需要一个 10 ms 的滴答(系统)计时器
//****************************************************************************
void
SysTickHandler(void)
{
    //调用 FatFs 滴答(系统)计时器
    disk_timerproc();
}
//****************************************************************************
//该函数执行"ls"命令。它可打开当前目录,详细显示其中内容,如文件属性、时间、日期、文件大
//小和名称等
//****************************************************************************
int
Cmd_ls(int argc, char *argv[])
{
    uint32_t ui32TotalSize;
    uint32_t ui32FileCount;
```

```
    uint32_t ui32DirCount;
    FRESULT iFResult;
    FatFS   * psFatFs;
    char    * pcFileName;
#if _USE_LFN
    char pucLfn[_MAX_LFN + 1];
    g_sFileInfo. lfname = pucLfn;
    g_sFileInfo. lfsize = sizeof(pucLfn);
#endif
    //打开当前要访问的目录
        iFResult = f_opendir(&g_sDirObject,g_pcCwdBuf);
    //检查错误,若有问题将返回
    if(iFResult != FR_OK)
    {
        return((int)iFResult);
    }
    ui32TotalSize = 0;
    ui32FileCount = 0;
    ui32DirCount = 0;
    //添加一空行(即换行)
    UARTprintf("\n");
    //通过循环来枚举目录中的所有条目
    for(;;)
    {
        //读取目录中的条目
        iFResult = f_readdir(&g_sDirObject,&g_sFileInfo);
        //检查错误,若有问题将返回
        if(iFResult != FR_OK)
        {
            return((int)iFResult);
        }
        //如果文件名为空,那么这就是列表的末尾
        //          //
        if(!g_sFileInfo. fname[0])
        {
            break;
        }
        //如果属性是目录,则将目录的个数 +1
        if(g_sFileInfo. fattrib & AM_DIR)
        {
            ui32DirCount ++;
        }
        //否则,它是一个文件夹。并使文件个数 +1,同时将文件大小添加到总计数值中
        else
        {
            ui32FileCount ++;
            ui32TotalSize += g_sFileInfo. fsize;
        }
#if _USE_LFN
```

```
            pcFileName = ( ( *g_sFileInfo.lfname) ? g_sFileInfo.lfname:g_sFileInfo.fname);
#else
            pcFileName = g_sFileInfo.fname;
#endif
            //用单独一行格式打印条目信息以显示:属性、日期、时间、大小和名称
            UARTprintf("%c%c%c%c%c %u/%02u/%02u %02u:%02u %9u  %s\n",
                    (g_sFileInfo.fattrib & AM_DIR) ? 'D' : '-',
                    (g_sFileInfo.fattrib & AM_RDO) ? 'R' : '-',
                    (g_sFileInfo.fattrib & AM_HID) ? 'H' : '-',
                    (g_sFileInfo.fattrib & AM_SYS) ? 'S' : '-',
                    (g_sFileInfo.fattrib & AM_ARC) ? 'A' : '-',
                    (g_sFileInfo.fdate >> 9) + 1980,
                    (g_sFileInfo.fdate >> 5) & 15,
                     g_sFileInfo.fdate & 31,
                    (g_sFileInfo.ftime >> 11),
                    (g_sFileInfo.ftime >> 5) & 63,
                     g_sFileInfo.fsize,
                    pcFileName);
        }
        //打印汇总行来显示文件、目录和大小的总数
        UARTprintf("\n%4u File(s),%10u bytes total\n%4u Dir(s)",
                ui32FileCount,ui32TotalSize,ui32DirCount);
        //获取可用空间
        iFResult = f_getfree("/",(DWORD *)&ui32TotalSize,&psFatFs);
        //检查错误,若有问题将返回
        if(iFResult != FR_OK)
        {
            return((int)iFResult);
        }
        //显示计算得出的可用空间大小
        UARTprintf(",%10uK bytes free\n",(ui32TotalSize *
                                    psFatFs->free_clust/2));
        //到了这里,没有错误则返回0。
        return(0);
}
//*****************************************************************************
//该函数执行“cd”命令。它需要一个参数来指定当前的工作目录
//路径分隔符必须使用正斜杠“/”。cd的参数可以是下列之一:
//*根目录("/")
//*指定完整的路径("/my/path/to/mydir")
//*当前目录的单个目录名("mydir")
//*父目录("..")
//不支持相对路径,所以不要尝试如下的操作:
//("../my/new/path")
//一旦指定了新目录,它会尝试打开该目录,以确保其存在
//如果新路径成功打开,则当前工作目录(CWD)改为新的路径
//*****************************************************************************
int
Cmd_cd(int argc,char *argv[])
```

```
{
    uint_fast8_t ui8Idx;
    FRESULT iFResult;
    //将当前工作路径复制到一个临时缓冲区中,以便对它进行操作
    strcpy(g_pcTmpBuf,g_pcCwdBuf);
    //如果第一个字符是"/",那么这是一个完全指定的路径,且它应该只当作"是"
    if(argv[1][0] =='/')
    {
        //确保新路径不大于 CWD 缓冲区
        if(strlen(argv[1]) +1 > sizeof(g_pcCwdBuf))
        {
            UARTprintf("Resulting path name is too long\n");
            return(0);
        }
        //如果新路径名(argv 中[1])不太长,那么将其复制到临时缓冲区, 以便
        //可对它进一步的检测
        else
        {
            strncpy(g_pcTmpBuf,argv[1],sizeof(g_pcTmpBuf));
        }
    }
    //如果参数是"..",将尝试删除 CWD 的最底层
    else if(!strcmp(argv[1],".."))
    {
        //在当前路径中得到最后一个字符的引索
        ui8Idx = strlen(g_pcTmpBuf) -1;
        //备份路径名的末尾,直到出现分隔符(/),或遇到路径的起始点为止
        while((g_pcTmpBuf[ui8Idx] !='/')&& (ui8Idx >1))
        {
            //备份一个字符
            ui8Idx --;
        }
        //现在要么在当前路径中的最底层分隔符(目录),要么在字符串的开头(根目录)
        //所以设置字符串新的结束位置,可有效地移除了路径的最后部分
        g_pcTmpBuf[ui8Idx] =0;
    }
    //否则,这只是一个来自当前目录的正常的路径名,并需要将其添加到当前路径中
    else
    {
        //测试添加到当前路径上新的附加路径,以确保对于完整的新路径有缓冲空间
        //它需要包括一个新的分隔符和结尾的空字符
        if(strlen(g_pcTmpBuf) +strlen(argv[1]) +1 +1 > sizeof(g_pcCwdBuf))
        {
            UARTprintf("Resulting path name is too long\n");
            return(0);
        }
        //新路径正确,所以添加分隔符,然后把新目录添加到路径中
        else
        {
```

```
            //如果尚未在根目录,那么添加一个分隔符“/”
            if(strcmp(g_pcTmpBuf,"/"))
            {
                strcat(g_pcTmpBuf,"/");
            }
            //添加新目录到路径中
            strcat(g_pcTmpBuf,argv[1]);
        }
    }
    //此时,一个待选的新目录路径在 chTmpBuf 中。尝试打开它来确保其有效性
    iFResult = f_opendir(&g_sDirObject,g_pcTmpBuf);
    //如果它不能被打开,那么这是一个条错误的路径,然后告知用户并返回
    if(iFResult != FR_OK)
    {
        UARTprintf("cd: %s\n",g_pcTmpBuf);
        return((int)iFResult);
    }
    //否则,它是一条有效的新路径,所以将其复制到 CWD 上
    else
    {
        strncpy(g_pcCwdBuf,g_pcTmpBuf,sizeof(g_pcCwdBuf));
    }
    //返回成功
    return(0);
}
//***************************************************************************
//此函数执行“PWD”命令。它简单地打印当前的工作目录
//***************************************************************************
int
Cmd_pwd(int argc,char *argv[])
{
    //打印 CWD 到控制台
    UARTprintf("%s\n",g_pcCwdBuf);
    //返回成功
    return(0);
}
//***************************************************************************
//此函数执行“cat”命令。它读取文件的内容,并将其输出到控制台上,这应该仅用于文本文件
//如果将其用于读取二进制文件,那么一堆无用输出可能会打印在控制台上
//***************************************************************************
int
Cmd_cat(int argc,char *argv[])
{
    FRESULT iFResult;
    uint32_t ui32BytesRead;
    //首先,判断文件名(即当前路径(CWD) + 文件名 + 一个分隔符和结尾空字符)能否存放到
    //临时缓冲区中
    if(strlen(g_pcCwdBuf) + strlen(argv[1]) + 1 + 1 > sizeof(g_pcTmpBuf))
    {
```

```
        UARTprintf("Resulting path name is too long\n");
        return(0);
    }
    //将当前路径复制到临时缓冲区中,以便可对它进行处理
    strcpy(g_pcTmpBuf,g_pcCwdBuf);
    //如果尚未在根目录下,则需添加一个分隔符
    if(strcmp("/",g_pcCwdBuf))
    {
        strcat(g_pcTmpBuf,"/");
    }
    //最后,添加的文件名被存放在一个完全指定的文件中
    strcat(g_pcTmpBuf,argv[1]);
    //读取打开的文件
    iFResult = f_open(&g_sFileObject,g_pcTmpBuf,FA_READ);
    //如果打开的文件中存在问题将返回一个错误
    if(iFResult != FR_OK)
    {
        return((int)iFResult);
    }
    //进入一个循环反复从文件中读取数据并显示它,直到读完文件为止
    do
    {
        //从文件中读取数据块
        iFResult = f_read(&g_sFileObject,g_pcTmpBuf,sizeof(g_pcTmpBuf) - 1,
                          &ui32BytesRead);
        //如果有错误读数,则打印一个换行,并返回错误给用户
        if(iFResult != FR_OK)
        {
            UARTprintf("\n");
            return((int)iFResult);
        }
        //0 作为读取最后一个块的终止符
        g_pcTmpBuf[ui32BytesRead] = 0;
        //打印接收到文件的最后一个块
        UARTprintf("%s",g_pcTmpBuf);
    }
    while(ui32BytesRead == sizeof(g_pcTmpBuf) - 1);
    //返回成功
    return(0);
}
//*****************************************************************************
//此函数执行“help”命令。打印可用的附简要说明的简单命令列表
//*****************************************************************************
int
Cmd_help(int argc,char *argv[])
{
    tCmdLineEntry *psEntry;
    //打印标题文本
    UARTprintf("\nAvailable commands\n");
```

```
    UARTprintf(" ---------------\n");
    //指向命令表的开始
    psEntry = &g_psCmdTable[0];
    //进入一个循环来读取命令表中的每个条目,当命令名为 NULL 时到达表的末尾
    while(psEntry -> pcCmd)
    {
        //打印命令名称和简要说明
        UARTprintf("%6s: %s\n",psEntry -> pcCmd,psEntry -> pcHelp);
        //前进到表中的下一个条目
        psEntry ++;
    }
    //返回成功
    return(0);
}
//*****************************************************************************
//该表保存命令的名称、执行函数和并简要描述
//*****************************************************************************
tCmdLineEntry g_psCmdTable[] =
{
    { "help",    Cmd_help,    "Display list of commands" },
    { "h",       Cmd_help,    "alias for help" },
    { "?",       Cmd_help,    "alias for help" },
    { "ls",      Cmd_ls,      "Display list of files" },
    { "chdir",   Cmd_cd,      "Change directory" },
    { "cd",      Cmd_cd,      "alias for chdir" },
    { "pwd",     Cmd_pwd,     "Show current working directory" },
    { "cat",     Cmd_cat,     "Show contents of a text file" },
    { 0,0,0 }
};
//*****************************************************************************
//如果驱动程序库遇到错误时调用的错误处理程序
//*****************************************************************************
#ifdef DEBUG
void
__error__(char *pcFilename,uint32_t ui32Line)
{
}
#endif
//*****************************************************************************
//配置 UART 和相关引脚。必须在调用 UARTprintf()之前完成配置
//*****************************************************************************
void
ConfigureUART(void)
{
    //使能用于 UART 的 GPIO 外设
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    //使能 UART0
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    //配置 GPIO 引脚为 UART 模式
```

```
    ROM_GPIOPinConfigure(GPIO_PA0_U0RX);
    ROM_GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE,GPIO_PIN_0 | GPIO_PIN_1);
    //使用内部 16MHz 振荡器作为 UART 的时钟源
    UARTClockSourceSet(UART0_BASE,UART_CLOCK_PIOSC);
    //初始化 UART 控制台 I/O
    UARTStdioConfig(0,115000,16000000);
}
//*****************************************************************************
//主函数。它执行初始化,然后运行命令处理循环从控制台读取命令
//*****************************************************************************
int
main(void)
{
    int nStatus;
    FRESULT iFResult;
    tRectangle sRect;
    //
    ROM_FPULazyStackingEnable();
    //设置系统时钟运行在 50 MHz
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                       SYSCTL_XTAL_16MHZ);
    //使能在本示例中使用的外设
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
    //设置系统定时器每 100 Hz 产生一个中断。该 FatFS 驱动需要一个 10 ms 的滴答时钟
    ROM_SysTickPeriodSet(ROM_SysCtlClockGet()/100);
    ROM_SysTickEnable();
    ROM_SysTickIntEnable();
    //使能中断
    ROM_IntMasterEnable();
    //初始化 UART 作为文本 I/O 控制台
    ConfigureUART();
    //初始化显示驱动程序
    CFAL96x64x16Init();
    //初始化图形上下文
    GrContextInit(&g_sContext,&g_sCFAL96x64x16);
    //在屏幕顶部创建一个蓝色条
    sRect.i16XMin = 0;
    sRect.i16YMin = 0;
    sRect.i16XMax = GrContextDpyWidthGet(&g_sContext) - 1;
    sRect.i16YMax = 9;
    GrContextForegroundSet(&g_sContext,ClrDarkBlue);
    GrRectFill(&g_sContext,&sRect);
    //改变前景色为白色文字
    GrContextForegroundSet(&g_sContext,ClrWhite);
    //把应用程序名放置在蓝色条的中间
    GrContextFontSet(&g_sContext,g_psFontFixed6x8);
    GrStringDrawCentered(&g_sContext,"sd_card", -1,
```

```
                        GrContextDpyWidthGet(&g_sContext)/2,4,0);
//在屏幕上显示指示
GrContextFontSet(&g_sContext,g_psFontFixed6x8);
GrStringDrawCentered(&g_sContext,"Connect a",-1,
                        GrContextDpyWidthGet(&g_sContext)/2,20,false);
GrStringDrawCentered(&g_sContext,"terminal",-1,
                        GrContextDpyWidthGet(&g_sContext)/2,30,false);
GrStringDrawCentered(&g_sContext,"to UART0.",-1,
                        GrContextDpyWidthGet(&g_sContext)/2,40,false);
GrStringDrawCentered(&g_sContext,"115200,N,8,1",-1,
                        GrContextDpyWidthGet(&g_sContext)/2,50,false);
//
//打印帮助信息给用户
UARTprintf("\n\nSD Card Example Program\n");
UARTprintf("Type \ help\ for help. \n");
//挂载文件系统,使用逻辑磁盘0
iFResult = f_mount(0,&g_sFatFs);
if(iFResult != FR_OK)
{
    UARTprintf("f_mount error: %s\n",StringFromFResult(iFResult));
    return(1);
}
//进入无限循环用于用户读取和发出处理命令
while(1)
{
    //打印提示到控制台。显示 CWD
    UARTprintf("\n%s>",g_pcCwdBuf);
    //从用户获取一行文本
    UARTgets(g_pcCmdBuf,sizeof(g_pcCmdBuf));
    //来自用户的命令行将被解析并有效的命令执行
    nStatus = CmdLineProcess(g_pcCmdBuf);
    //错误命令的处理情况
    if(nStatus == CMDLINE_BAD_CMD)
    {
        UARTprintf("Bad command!\n");
    }
    //参数太多的处理情况
    else if(nStatus == CMDLINE_TOO_MANY_ARGS)
    {
        UARTprintf("Too many arguments for command processor!\n");
    }
    //否则,该命令执行。如果退回则打印错误代码
    //          //
    //
    else if(nStatus != 0)
    {
        UARTprintf("Command returned error code %s\n",
                     StringFromFResult((FRESULT)nStatus));
    }
```

```
    }
}
```

2）导入 sd_card 工程，如图 15-17 所示。在 C:\ti\TivaWare_C_Series－2.0.1.11577\examples\boards\dk－tm4c123g\sd_card 目录下单击 sd card 图标导入 sd_card 工程，如图 15-19 所示。

3）将 C:\ti\TivaWare_C_Series－2.0.1.11577\third_party\fatfs\src\option 目录下支持简体中文的文件（cc936.c）添加到工程中。

4）编译工程生成可执行的.axf 文件，如图 15-20 所示。

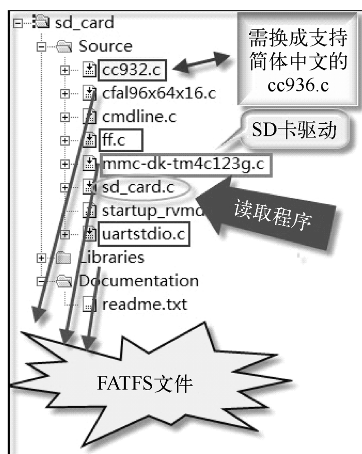


图 15-19　sd_card 工程

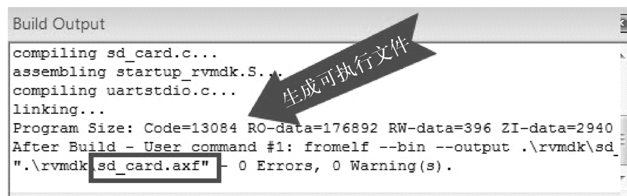


图 15-20　sd_card 工程编译结果

5）制作测试用的几个文件。

① 在读卡器中把 microSD 卡格式化为 FAT32 格式，按如图 15-21 所示设置。

② 制作文本文件和文件夹，如图 15-22 所示。



图 15-21　将 microSD 卡格式化成 FAT32 格式



图 15-22　制作两个空文件夹和一个文本文件

6）在 BD－LM4F232 评估板与 PuTTY 中测试程序。

① 将.axf 文件下载到 BD－LM4F232 评估板中。

② 打开 PuTTY，在按下板上的重启按钮后，如图 15-23 所示。

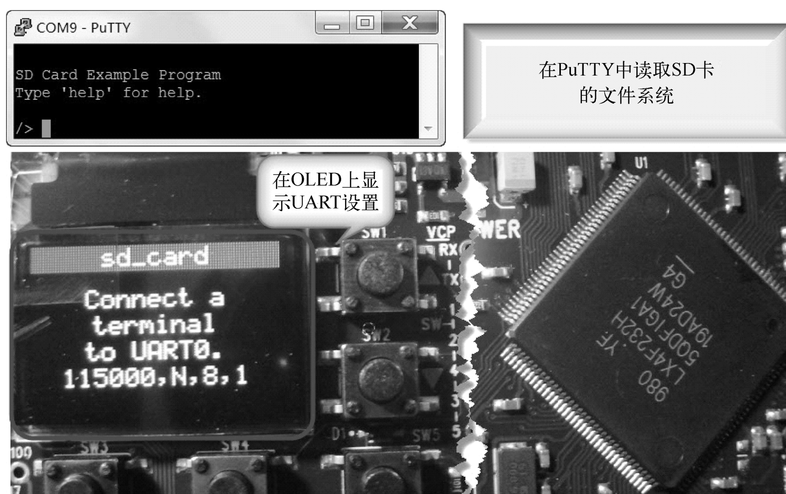


图 15-23　在 PuTTY 中读取 FatFS

③ 在提示符处输入 help 命令，如图 15-24 所示。

④ 输入 ls 命令显示卡中的文件，如图 15-25 所示。

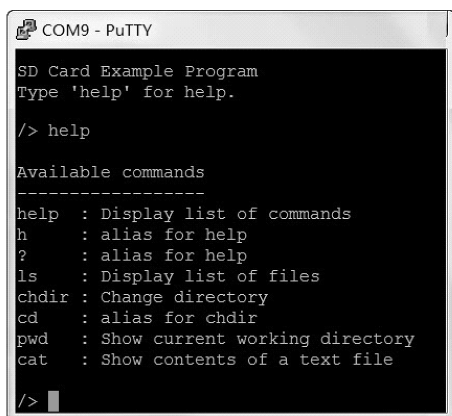


图 15-24　显示 help 菜单

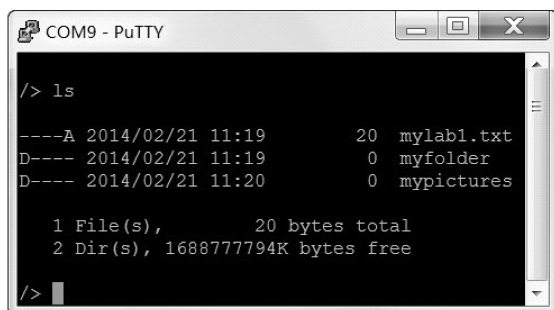


图 15-25　输入 ls 显示卡中的文件

⑤ 输入 cat 命令显示文本文件的内容，如图 15-26 所示。



图 15-26　显示文本文件的内容

为了控制篇幅，文件系统的测试到此为止，有兴趣的读者可以继续测试其他功能，或自行修改上述代码增加其他功能。

第 16 章

基本图形库（Grlib）

本章将扼要介绍液晶显示屏的基本特点、TivaWare 图形库与基于图形库的应用（包括介绍了 OLED 液晶屏底层驱动程序的写法）。

本章的主要内容：

- 图形库介绍
- TivaWare 图形库简介
- 例程



16.1 图形库与液晶屏概述

16.1.1 图形库概述

TivaWare 图形库是一组免版税的图形基元和小工具集，用于在基于 Tiva C 系列微处理器的具有图形显示的电路板上创建图形用户界面。该图形库包含三个功能构建层：显示屏驱动器层、图形基元层、小工具层。

1. 显示屏驱动器层

基于使用中的显示屏，该层的主要任务是完成下列与显示硬件相关的低层接口程序：

1）显示相关的操作程序。

① 初始化。

② 背光控制。

③ 对比度。

④ 根据彩色图将 24 位 RGB 值转换到屏幕上。

2）绘制图形库程序。

① 刷新。

② 画线。

③ 画像素点。

④ 矩形绘制。

3）需用户修改的硬件相关程序。

① 将显示器与 TM4 C 器件相连。

② 根据使用的显示设备编写或修改现存的驱动程序（如颜色深度和大小）。

2. 图形基元层

该层为低层绘图支持，具有以下特点：

1）可绘制点、线、矩形、圆、字体、位图图形和文本。

2）支持离屏缓冲。

3）绘图上下文的前景和背景。

4）由 24 位的 RGB 值表示一种颜色（每色 8 位）。预先定义 150 种颜色。

5）预先定义 153 个基于现代计算机的字体。

3. 小工具层

它提供复选框、按钮、单选按钮、滑块、列表框以及一个或多个图像基元的通用封装，以便在显示屏上绘制用户界面元素，并能够通过小工具元素为用户交互提供应用程序定义的响应，如图 16-1 所示。

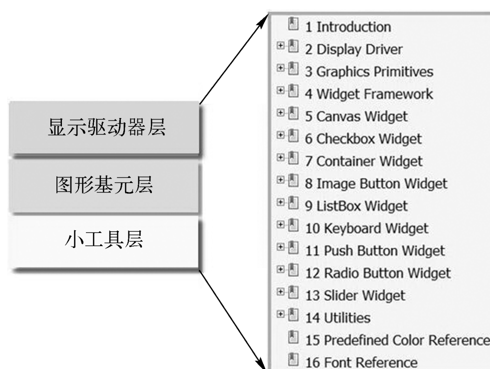


图 16-1　图形库框架

为了确保该软件易于理解和维护，Tiva C 图形库完全采用 C 语言编写（除无法实现的以外）。虽然用 C 语言编写，但由于 Cortex - M4 Thumb2 指令集的紧凑性，该库在存储和处理器使用方面仍然非常有效。其特点如下：

① 免费许可证和免版税使用权（与 TI ARM Cortex MCU 配合使用）。

② 简化并加快应用程序的开发，可用于应用程序开发或作为编程示例。

③ 可创建功能完整、易于维护的代码。

④ 完全利用 Cortex - M4 内核的星形中断性能，无需任何特殊的 pragma 或自定义汇编语言代码的起始代码/结束代码功能。

⑤ 可使用错误检查代码进行编译（用于开发），也可不使用（用于具有较小存储器配置的 MCU 中的最终生产）。

⑥ 可作为对象库和源代码，以便按原样使用该库或根据需要修改。

⑦ 每个外设的完整源代码示例可用于所有 Tiva C 开发和评估套件的完整项目。

⑧ 在 CCS5. x、ARM/Keil、IAR、Code Red、CodeSourcery 以及通用 GNU 开发工具上编译。

16. 1. 2 液晶屏简介

目前在国内市场上常见的液晶显示器主要有两种：LCD 和 OLED。

1. LCD 液晶显示器

① STN－LCD（Super Twisted Nematic）：超扭转向列式液晶。

② LTPS（Low TemperaturePoly Silicon）：低温多晶硅。

③ TFT－LCD（Thin Film Transistor－Liquid Crystal Display）：薄膜晶体管液晶显示器。

而以 TFT－LCD 应用最广泛，它由光导、偏振片、玻璃基板、储能电容、源驱动（数据）、门驱动、TFT、公共电极、滤色片等组成，其特点是在显示屏的每一个像素点上都有一个薄膜晶体管，使每个像素都可以通过脉冲信号来直接控制。可有效克服非选通时的串扰，使显示的图形色彩鲜艳逼真。尽管液晶本身并不发光，但可通过对液晶施加不同的电压来改变液晶分子的排列顺序，结合偏振片与 RGB 彩色滤光片等控制机构，就可将背光源的光信号（包含要显示图形的信息）打在液晶面板上，从而实现图形的显示，如图 16-2～16-5 所示。

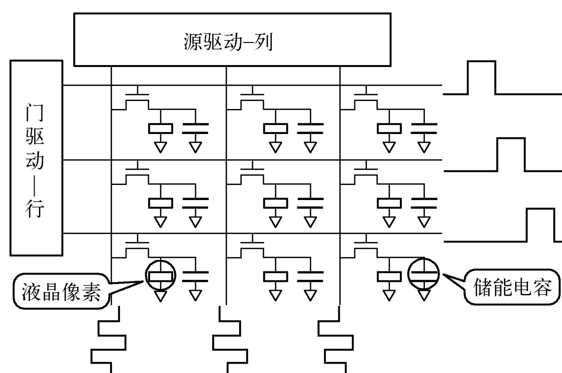


图 16-2　TFT－LCD 的等效电路

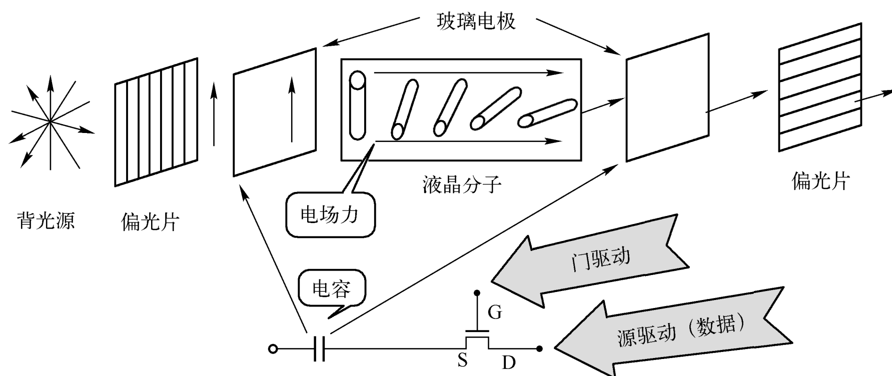


图 16-3　TFT 亮度控制示意图

从图 16-5 中可以看到，要使 TFT－LCD 正常工作，需要一个控制系统来协同 TFT 各个模块之间的关系，因此各种控制芯片应运而生。市面上常用的控制芯片有 ILI9xxx 系列（比如 ILI9320、ILI9341 等）和 TI 公司制作的开发板上常用的 SSD2119 等。

（1）数据显示格式

常用的数据显示格式有 18BPP 的 RGB666（占 3 个字节）与 16BPP 的 RGB565（占 2 个

字节）格式，如图 16-6 所示。

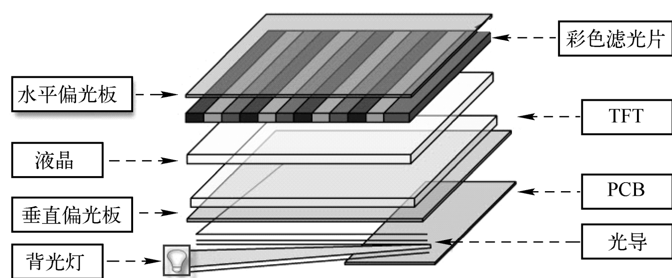


图 16-4　TFT - LCD 液晶屏组件示意图

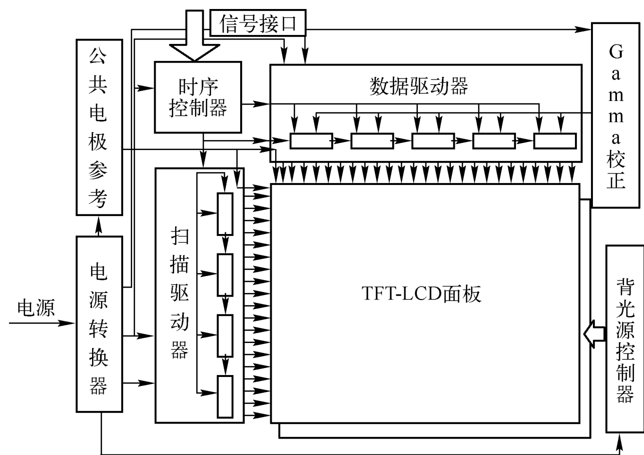
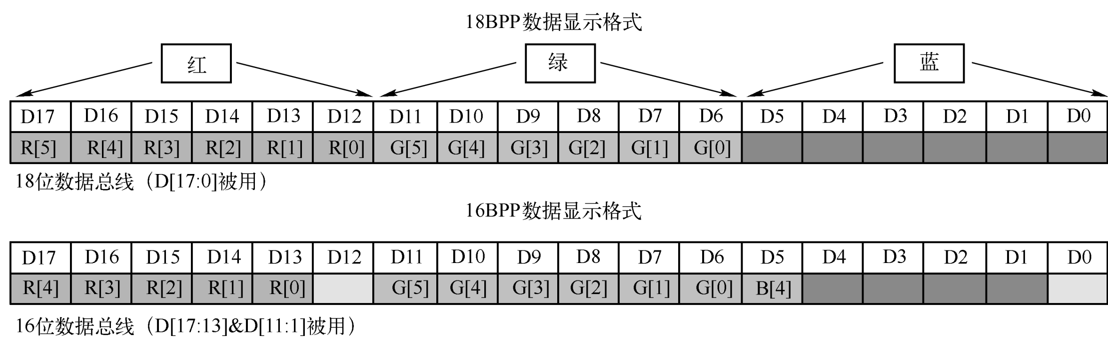


图 16-5　TFT - LCD 驱动框图



18BPP数据显示格式

红 | 绿 | 蓝

D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
R[5]	R[4]	R[3]	R[2]	R[1]	R[0]	G[5]	G[4]	G[3]	G[2]	G[1]	G[0]						

18位数据总线（D[17:0]被用）

16BPP数据显示格式

D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
R[4]	R[3]	R[2]	R[1]	R[0]		G[5]	G[4]	G[3]	G[2]	G[1]	G[0]	B[4]					

16位数据总线（D[17:13]&D[11:1]被用）

图 16-6　TFT - LCD 常用的显示格式

（2）ILI9320 常用的几个命令

① 索引寄存器（IR），如图 16-7 所示。

R/W	RS	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
W	0	–	–	–	–	–	–	–	–	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0

图 16-7　索引寄存器

该索引寄存器指定寄存器（R00h～RFFh）的地址或将要访问的RAM。

② 开始振荡命令（R00h），如图16-8所示。

R/W	RS	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
W	1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	1
R	1	1	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0

图16-8　开始振荡寄存器

通过设置OSC位为“1”来启动内部振荡器；而设置“0”时停止振荡器工作。至少等待10 ms使振荡器的频率稳定后，方可进行其他功能的设置。当对该寄存器进行读取操作时，将返回器件型号为“9320”h。

③ 入口模式命令（R03h），如图16-9上部所示。

R/W	RS	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
W	1	TRI	DFM	0	BGR	0	0	HWM	0	ORG	0	I/D1	I/D0	AM	0	0	0

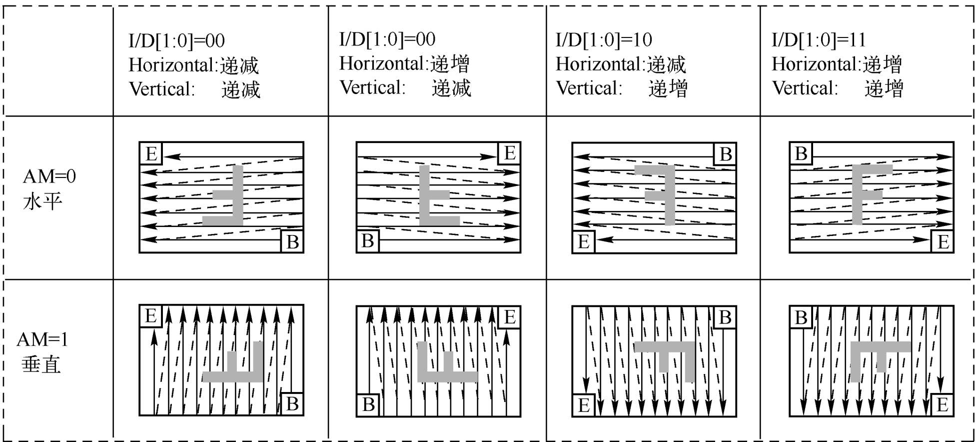


图16-9　入口模式寄存器

从图16-9中可看到，AM用于GRAM更新方向的控制。当AM =“0”，水平写方向的地址得到更新；而当AM =“1”时，垂直写方向的地址得到更新。当用寄存器R16h和R17H设置了一个窗口区域，仅基于I/D[1:0]的GRAM寻址区域和AM置位的区域得到更新。

当更新一个显示数据的像素，I/D[1:0]控制的地址计数器（AC）会自动进行增1或减1的操作，详细过程如图16-9下部所示。

④ 显示控制1命令，如图16-10所示。

R/W	RS	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
W	1	0	0	PTDE1	PTDE0	0	0	BASEE	0	0	0	GON	DTE	CL	0	D1	D0

图16-10　显示控制1命令

CL：用来控制是8位色，还是262K色。当CL为0时262K色，当CL为1时8位色。D1、D0、BASEE这3位用于显示器的关断控制。当全部置1时打开显示；当其全0时关断

显示。可应用该命令来使能和禁止显示，合理使用电能。

D[1:0]：设置 D[1:0]=“11”来打开显示面板；设置 D[1:0]=“00”关闭显示面板。

当写 D1=“1” 图形将显示在面板上；而写 D1=0 停止图形显示。

当写 D1=“0”图形显示数据将被保留在内部 GRAM 中；而当写 D1=“1”时，ILI9320 将显示图形数据。

当 D1=“0”时，面板上将不会有显示，所有的源输出将接地，以减少驱动液晶时所需的充电/放电电流。

虽然可通过设置 D[1:0]=“01”来关闭显示，但 ILI9320 仍会继续进行内部的显示操作。而当设置 D[1:0]=“00”时，却可以完全关闭显示器工作（这时 ILI9320 的内部显示操作将全部停止）。结合 GON 与 DTE 位的设置，可通过设置 D[1:0]来控制显示的关/断。

该命令 CL 位用来控制是 8 位色，还是 262K 色。

BASEE：基本图像显示使能位。当 BASEE=“0”时，禁止基本图像显示。ILI9320 驱动液晶在不点亮显示等级或者仅显示部分图像。当 BASEE=“1”时，使能基本图像显示。D[1:0]设置的优先级高于 BASEE 的设置。

PTDE[1:0]：部分图像 2 和部分图像 1 的使能位。当 PTDE1/0=“ 0”时，关闭部分图像，仅显示基本图像；而但 PTDE1/0 =“1”时，使能部分图像显示，且需设置基本图像显示使能位为 0（BASEE=0）。

显示控制 1 命令中的位与其实现的功能或操作，如图 16-11 所示。

①

D1	D0	BASEE	Source, VCOM Output	IL19320 internal operation
0	0	0	GND	停止
0	1	1	GND	操作
1	0	0	不点亮显示	操作
1	1	0	不点亮显示	操作
1	1	1	基本图形显示	操作

②

CL	Colors
0	262,144
1	8

③

GON	DTE	G1~G320 Gate Output
0	0	VGH
0	1	VGH
1	0	VGL
1	1	正常显示

图 16-11　显示控制命令中的位与其相应操作的关系

⑤ GRAM 水平/垂直地址设置（R20h，R21h）。AD[16:0]设置地址计数器（AC）的初始值。址计数器（AC）可根据 AM 的设置自动更新，I/D 位作为被写入到内部 GRAM 的数据。当从内部的 GRAM 中读取数据时，地址计数器不会自动更新。如图 16-12 所示。

⑥ 写数据到 GRAM 命令（R22h），如图 16-13 所示。该寄存器是 GRAMd 的访问端口。当通过该寄存器更新显示数据时，地址计数器（AC）自动递增/递减。

⑦ 水平和垂直 RAM 的地址位置设置（R50h、R51h、R52h、R53h），如图 16-14 所示。

R/W	RS		D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
W	1		0	0	0	0	0	0	0	0	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
W	1		0	0	0	0	0	0	0	AD16	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8

AD[16:0]	GRAM Data Map
17'h00000 ~ 17'h000EF	1st line GRAM Data
17'h00100 ~ 17'h001EF	2nd line GRAM Data
17'h00200 ~ 17'h002EF	3rd line GRAM Data
17'h00300 ~ 17'h003EF	4th line GRAM Data
17'h13D00 ~ 17'h13DEF	318th line GRAM Data
17'h13E00 ~ 17'h13EEF	319th line GRAM Data
17'h13F00 ~ 17'h13FEF	320th line GRAM Data

图 16-12　GRAM 水平/垂直地址设置

R/W	RS	D17 D16 D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
W	0	向RAM写入数据(WD[17:0]，在DB[17:0]为每个接口分配不同的引脚

图 16-13　写数据到 GRAM 命令

	R/W	RS	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
R50h	W	1	0	0	0	0	0	0	0	0	HSA7	HSA6	HSA5	HSA4	HSA3	HSA2	HSA1	HSA0
R51h	W	1	0	0	0	0	0	0	0	0	HEA7	HEA6	HEA5	HEA4	HEA3	HEA2	HEA1	HEA0
R52h	W	1	0	0	0	0	0	0	0	VSA8	VSA7	VSA6	VSA5	VSA4	VSA3	VSA2	VSA1	VSA0
R53h	W	1	0	0	0	0	0	0	0	VEA8	VEA7	VEA6	VEA5	VEA4	VEA3	VEA2	VEA1	VEA0

图 16-14　水平和垂直 RAM 的地址位置设置

HSA[7:0]/HEA[7:0]：水平方向窗口的起始地址和结束地址。通过设置 HSA 和 HEA 位可调整写数据时 GRAM 水平区域的大小。设置 HSA 和 HEA 位必须在开始写 RAM 操作之前进行，且必须保证："00"h ≤HSA[7:0] < HEA[7:0]≤"EF"h（即 239），以及"04"h ≦ HEA - HAS。

VSA[8:0]/VEA[8:0]：垂直方向窗口的起始地址和结束地址。通过设置 VSA 和 VEA 位可调整写数据时 GRAM 垂直区域的大小。同样，设置 VSA 和 VEA 位必须在开始写 RAM 操作之前进行，且必须保证："00"h≤VSA[8:0] < VEA[8:0]≤"13F"h（即 319）。这几个命令可用于设置自定义显示区域的大小，如图 16-15 所示。

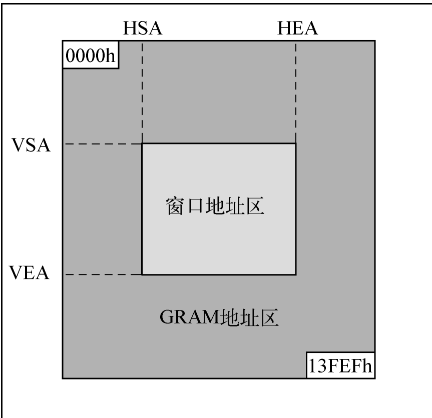


图 16-15　设置的显示区域大小

2. OLED 液晶显示屏

OLED（Organic Light - Emitting Diode）为有机发光二极管液晶显示屏。其发光原理：当从阴极注入的电子和阳极注入的空穴在电场力的作用下，这两种载流子相向而行，在有机材料中由于相遇复合而释放能量，从而导致有机发光材料分子中的原子核外围的电子从低能态跃迁到高能态，当

这些高能电子返回低能态时将会出现发光现象。

驱动 OLED 中的一个像素点可分为三个阶段：在第 1 阶段，段驱动首先把像素复位到低电平，以便释放存储在段电极周边寄生电容中先前数据的电荷，并且可通过命令（编程）来缩短放电的时间；在第 2 阶段，段驱动把 A、B 或 C 色彩中的像素分别预充电到所需的电平值 VPA、VPB 和 VPC，同样可通过命令（编程）来缩短充电的时间；最后阶段是电流驱动阶段，采用 PWM 方式，通过段驱动中的电流源给像素点提供恒定的驱动电流。OLED 段和公共驱动模块框图与驱动信号波形分别如图 16-16、16-17 所示。

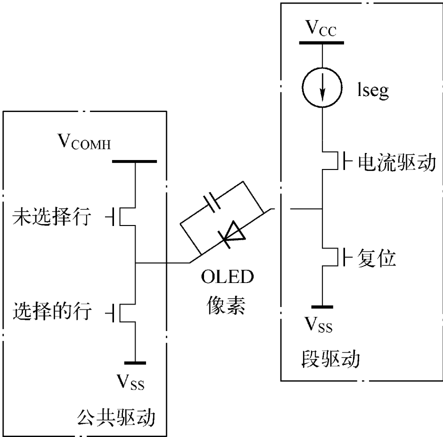


图 16-16　OLED 段和公共驱动模块框图

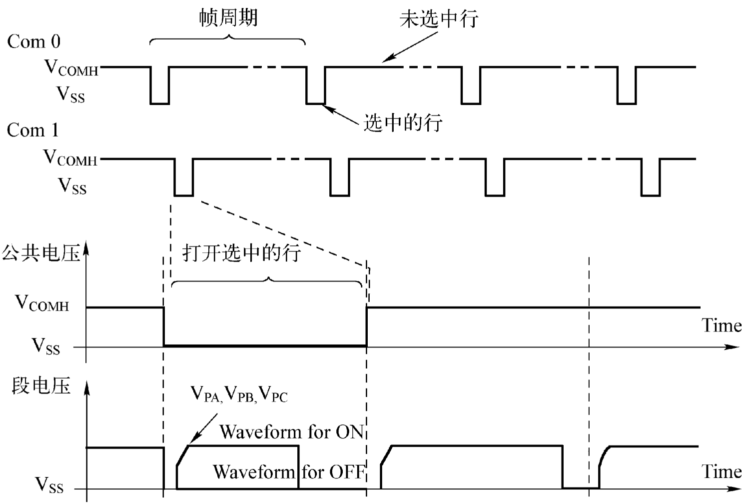


图 16-17　段和公共驱动信号的波形

在图 16-16 与 16-17 中可以看到，段驱动器有 288（96 ×3）个电流源驱动 OLED 面板（选用 96 ×64 OLED）。通过对比度设置命令，可将驱动电流在 0 ~ 200 μA 内进行调整，总共 256 级。公共驱动器会产生电压扫描脉冲，并进行逐行顺序扫描。如果没有行被选中，该行的所有像素将被驱动到电压 VCOMH 的反相偏置，在扫描行时，该行的所有像素可通过发送相应的数据信号到对应段引脚来打开或关闭这些像素点。如果该像素点被关闭，段驱动电流将保持 0 电平。反之，该像素点打开，段驱动电流为 I_{SEG}。

（1）OLED 的主要优点

其主要优点有质轻、厚度薄、可弯曲、省电、自发光、对比度较高、视角广、反应速度快等。由于 OLED 比 TFT - LCD 屏性能更为优越，已广泛地应用于各种电子产品中。下面将介绍由 SSD1332 控制芯片和 96 ×64 面板组成的 OLED 液晶。

（2）SSD1332 控制芯片的模块框图

SSD1332 控制芯片的模块框图如图 16-18 所示。

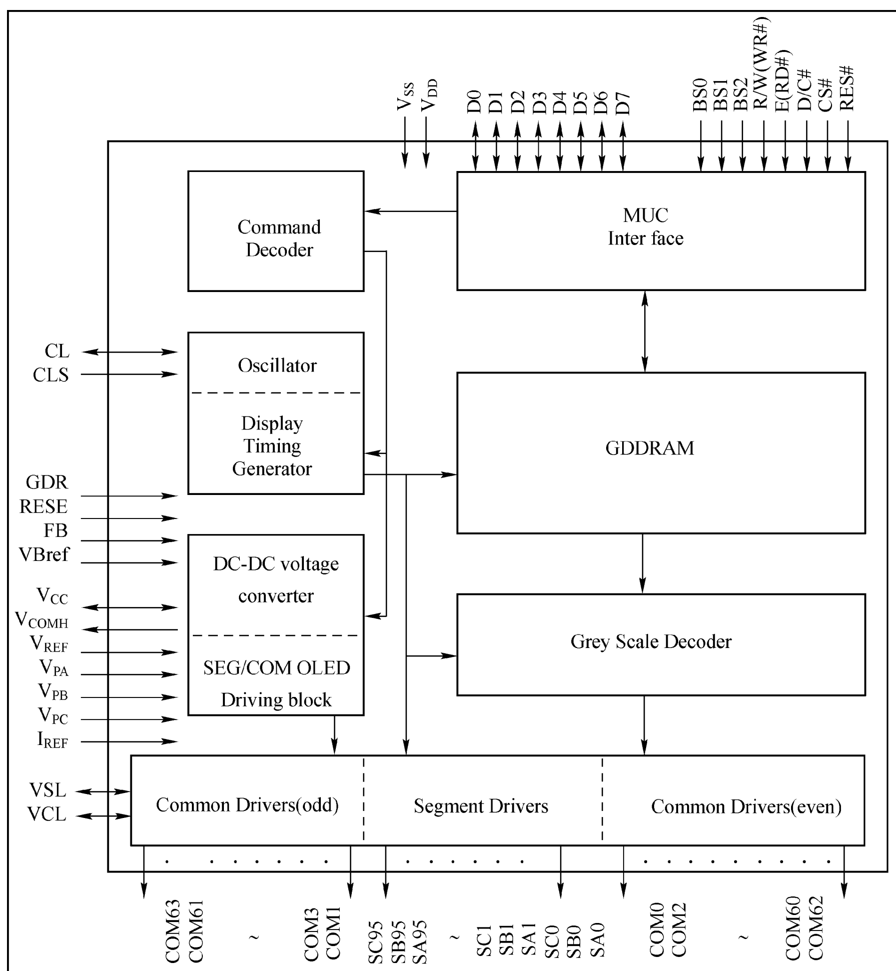


图 16-18　SSD1332 控制芯片的模块框图

（3）通信方式采用串行（SPI）模式

其时序如图 16-19 所示。图中信息描述见表 16-1。

表 16-1　图中信号描述

信号名称	描　述	信号名称	描　述
D/C#	数据/命令标志（0：命令；1：数据）	CS#	片选信号
SCLK（D_0）	串行时钟线	SDIN（ D_1）	串行数据线
t_{cycle}	时钟周期	t_{DSW}	写数据建立时间
t_{AS}	地址建立时间	t_{DHW}	写数据保持时间
t_{AH}	地址保持时间	t_{CLKL}	时钟低电平时间
t_{CSS}	片选建立时间	t_{CLKH}	时钟高电平时间
t_{CSH}	片选保持时间	t_R	上升时间
t_F	下降时间		

从图 16-19 中可以看到，当 CS#为低电平时，将使能 SSD1332 控制器。在每个 SCLK 号的上升沿，SDIN 上的数据会以 D7、D6……D0 的次序移入到 8 位移位寄存器中。每 8 个时

钟周期采样 1 次，如果 D/C#信号置高电平，则移位寄存器中的数据将会写入 GDDRAM；如果 D/C#信号置低电平，则移位寄存器中的数据将会写入指令寄存器。

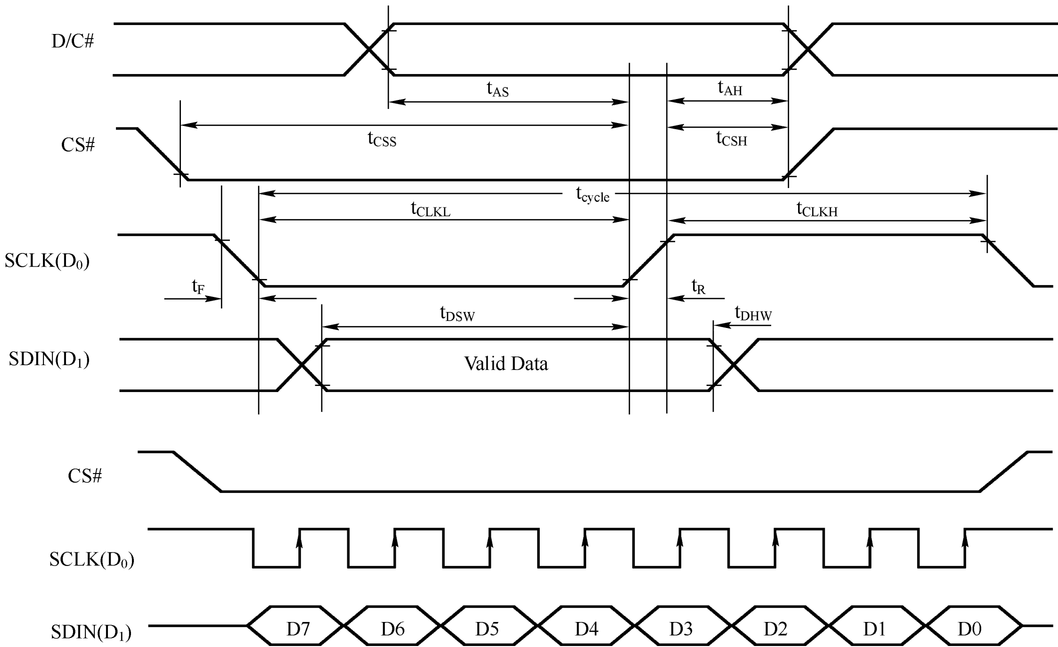


图 16-19　串行接口的时序图

（4）图形显示数据 RAM（GDDRAM）

GDDRAM 为要显示图形的位映射静态 RAM 保存，其大小为 96 × 64 × 16 位，且操作灵活，可通过软件来实现段和公共输出的重映射。

对于显示的垂直滚动，内部寄存器存储显示的起始行可以被设置成控制 RAM 数据的部分被映射到显示。每个像素点具有 16 位的数据，颜色 A、B 和 C 三个子像素分别为 6 位、5 位和 6 位。数据像素的图形显示数据 RAM 结构如图 16-20 所示。

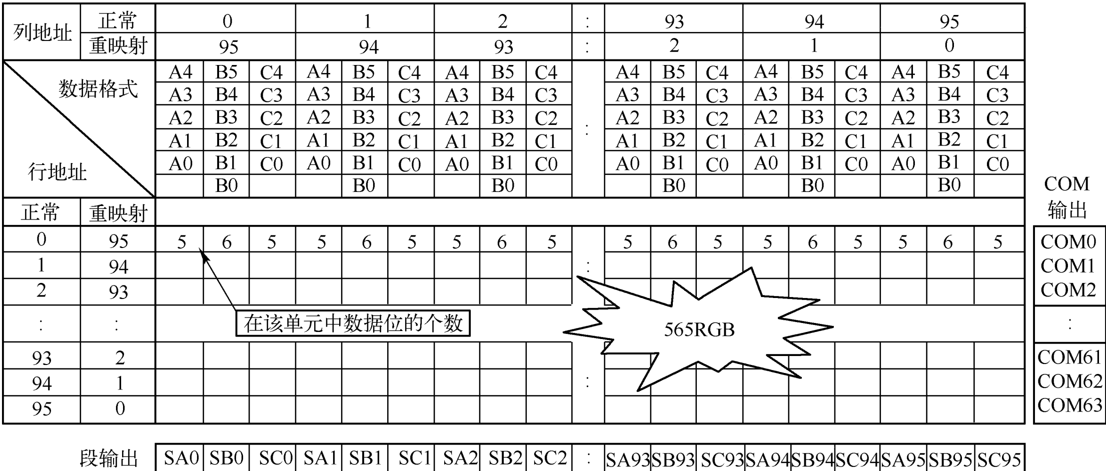


图 16-20　65K 色彩深度图形显示数据 RAM 结构

发送一个 16 位数据的像素被分成两个 8 位数据的传输，如图 16-21 所示。

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1st字节	C4	C3	C2	C1	C0	B5	B4	B3
2nd字节	B2	B1	B0	A4	A3	A2	A1	A0

图 16-21　65K 色彩深度图形显示数据写入序列

在 256 色模式中，每个像素由 8 位构成。颜色 A 用 2 位表示，而颜色 B 和 C 由 3 位表示。虽然只用 8 位来表示一个像素，但每个像素在内部图形显示数据 RAM 中却要占用 16 位存储空间，其格式如图 16-22 所示。

	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1st 字节	C2	C1	C0	B2	B1	B0	A1	A0

C (3 bits)	RAW (5 bits)
000	00000
001	00100
010	01000
011	01100
100	10010
101	10110
110	11010
111	11110

B (3 bits)	RAM 内容 (6 bits)
000	000000
001	001000
010	010000
011	011000
100	100100
101	101100
110	110100
111	111100

A (2 bits)	RAM (5 bits)
00	00000
01	01000
10	10100
11	11100

1个像素点

565RGB

图 16-22　一个像素 256 色深度的图形显示数据 RAM 结构

（5）OLED 的几个常用命令

OLED 的几个常用命令的功能介绍见表 16-2。

表 16-2　几个常用命令的功能介绍（MCU 接口引脚设置为 D/C＝0、R/W(WR#)＝0、E(RD#)＝1）

D/C	Hex	D7	D6	D5	D4	D3	D2	D1	D0	命　令	描　述
0 0 0	15 A[6:0] B[6:0]	0 * *	0 A6 B6	0 A5 B5	1 A4 B4	0 A3 B3	1 A2 B2	0 A1 B1	1 A0 B0	设置列地址	A[6:0]设置列地址从 0～95，复位＝00d B[6:0]设置列结束地址从 0～95 复位＝95d
0 0 0	75 A[5:0] B[5:0]	0 * *	1 * *	1 A5 B5	1 A4 B4	0 A3 B3	1 A2 B2	0 A1 B1	1 A0 B0	设置行地址	A[5:0]设置行起始地址 0～63，复位＝00d B[5:0]设置行结束地址 0～63，复位＝63d
0 0	81 A[7:0]	1 A7	0 A6	0 A5	0 A4	0 A3	0 A2	0 A1	1 A0	设置颜色 A 的对比度 （段引脚：SA0～SA95）	双字节命令从 1～256 个对比度步长可选 对比度随步数的增加而增大，复位＝80 h
0 0	62 A[7:0]	1 A7	0 A6	0 A5	0 A4	0 A3	0 A2	1 A1	0 A0	设置颜色 B 的对比度 （段引脚：SB0～SB95）	双字节命令从 1～256 个对比度步长可选 对比度随步数的增加而增大，复位＝80 h
0 0	83 A[7:0]	1 A7	0 A6	0 A5	0 A4	0 A3	0 A2	1 A1	1 A0	设置颜色 C 的对比度 （段引脚：SC0～SC95）	双字节命令从 1～256 个对比度步长可选 对比度随步数的增加而增大，复位＝80 h
0	A[3:0]	*	*	*	*	A3	A2	A1	A0	主电流控制	设置 A[3:0]从 0000，0001…到 1111 来调节主电流衰减因子从 1/16，2/16…到 16/16。复位＝1111b，无衰减

（续）

D/C	Hex	D7	D6	D5	D4	D3	D2	D1	D0	命　　令	描　　述
0 0	A0 A[7:0]	1 A7	0 A6	1 A5	0 A4	0 *	0 *	0 A1	0 A0	设置重映射和数据格式	A[0]=0，水平地址增量（复位） A[0]=1，垂直地址增量 A[1]=0，列地址0映射到SEG0（复位） A[1]=1，列地址95映射到SEG0列地址 A[4]=0，从COM 0扫描到COM（N-1） A[4]=1，从COM（n-1）扫描到COM0。其中N是复用率 A[5]=0，禁止把COM拆分成奇偶（复位） A[5]=1，使能把COM拆分成奇偶 A[7:6]=00；256彩色格式 =01；65 k彩色格式（复位）
0 0	A1 A[5:0]	1 *	0 *	1 A5	0 A4	0 A3	0 A2	0 A1	1 A0	设置显示起始行	设置GRAM的显示起始行寄存器从0-63 显示起始行寄存器复位后重置为00h
0 0	A2 A[5:0]	1 *	0 *	1 A5	0 A4	0 A3	0 A2	1 A1	0 A0	设置显示偏移量	由COM设置从0~63垂直滚动 该值复位后重置为00h
0	A4~A7	1	0	1	0	0	1	X1	X0	设置显示模式	A4h=正常显示（复位） A5h=开启全屏显示，所有像素在GS 63级 A6h=关闭全屏显示，关闭所有的像素 A7h=逆显示
0	AE~AF	1	0	1	0	X3	1	1	1	显示开/关	AEh=显示关闭（复位） AFh=打开显示



16.2　TivaWare 图形库简介

由于TivaWare 2.0版图形库的数据手册近300页，为了控制本书的篇幅，本小节仅就图形库作一扼要介绍，并对几个常用库函数的功能进行说明。

16.2.1　图形库的特点

（1）主要特点

1）免费许可证和免版税使用权（与Tiva C MCU配合使用）。

2）简化并加快应用程序的开发，可用于应用程序开发或作为编程示例。

3）可创建功能完整、易于维护的代码。

4）Tiva图形库完全采用C语言编写（除了不可能实现的例外）。

5）完全利用Cortex-M4内核的中断性能，无需任何特殊的progma或自定义汇编起始代码/结束代码功能。

6）既可使用错误检查代码进行编译（用于开发），也可不使用（用于具有较小存储器

配置的 MCU 中的最终生产）。

7）可作为对象库和源代码，以便按原样使用图形库或根据需要进行修改。

8）每个外设的完整源代码示例可用于所有 Tiva 开发和评估套件的完整项目。

9）可在 CCS4/5/6、ARM/Keil、IAR、Code Red、CodeSourcery 以及通用 GNU 开发工具上编译所建立的图形库工程。

（2）特殊工具（utilities）

下列工具用于产生与图形库兼容的数据结构。

1）ftrasterize。

① 使用 FreeType 字体字体渲染包将自定义字体转换成图形库格式。

② 支持的字体包括：OpenType 字体的 TrueType®、PostScript®Type 1 和 Windows®FNT。

2）lmi - button。使用脚本插件 GIMP 创建自定义形状按钮。产生的图像可用于按钮控件。

3）pnmtoc。

① 将 NetPBM 图像文件转换成一个图形库兼容的文件格式。

② NetPBM 图像格式可由 GIMP、NetPBM、ImageMajik 等产生。

16.2.2 图形库源代码

表 16-3 是图形库源代码的组织概述及各部分的功能简介。

表 16-3 源代码简介

程序名称	功能描述
Makefile	构建图形库的规则
canvas. c	画布的源代码
canvas. h	画布的头文件
ccs/	Code ComposerStudio 工程文件目录
charmap. c	程序代码页映射函数的源代码
checkbox. c	复选框的源代码
checkbox. h	复选框的头文件
circle. c	圆基元的源代码
container. c	容器控件的源代码
container. h	容器控件的头文件
context. c	绘图上下文的源代码
fonts/	图形库提供的包含字体结构的字体源文件
ftrasterize/	ftrasterize 程序的源代码
grlib. Opt	用于构建图形库的 KeiluVision 工程选项文件
grlib. Uv2	用于构建图形库的 KeiluVision 工程文件
grlib. ewd	用于构建图形库的 IAR Embedded Workbench 工程选项文件
grlib. ewp	用于构建图形库的 IAR Embedded Workbench 工程文件
grlib. h	包含基本图形原型的头文件

（续）

程序名称	功能描述
image. c	基本图形的源代码
line. c	线基元的源代码
offscr1bpp. c	1BPP 离屏缓冲显示驱动程序源代码
offscr4bpp. c	4BPP 离屏缓冲显示驱动程序的源代码
offscr8bpp. c	8BPP 离屏缓冲显示驱动程序的源代码
pnmtoc/	pnmtoc 程序的源代码
pushbutton. c	按钮控件的源代码
pushbutton. h	按钮控件的头文件
radiobutton. c	单选按钮的源代码
radiobutton. h	单选按钮的头文件
readme. txt	图形库高亮的简短描述文件
rectangle. c	矩形基元的源代码
slider. c	滑动控件的源代码
slider. h	滑动控件的头文件
string. c	字符串基元的源代码
widget. c	控件框架的源代码
widget. h	包含控件框架原型的头文件

16. 2. 3 图形固件库函数

1. 显示驱动库

（1）显示驱动库概述

显示驱动用于图形库与特定显示器的接口。它负责处理有关显示的底层细节，包括与显示控制器进行通信和理解显示控制器所需命令的行为。

显示驱动程序必须提供两件事情；通过图形库将所需程序集绘制到屏幕上和一组用于执行与显示操作有关的例程。显示相关的操作会根据不同显示器有所不同，但都包含初始化程序，并可能包括诸如背光控制和对比度控制。

由图形库所需的例程组成的一个结构体来描述图形库的显示驱动程序。tDisplay 结构体包含这些函数指针，以及屏幕的宽度和高度。显示驱动程序提供了这种结构的实例，以及在特定显示驱动程序的头文件中定义该结构体的原型。

对于一些显示器，它可能在本地存储器的缓冲区中绘制更有效，并在所有的绘图操作完成后将结果复制到屏幕上。这就是常用的 4BPP 真彩显示，其中在显示存储器中的每个字节有 2 个像素。在这种情况下，Flush( ) 操作用于指示应该被复制到显示器上的本地显示缓冲区。

（2）tDisplay 结构体定义

该结构体定义了与显示驱动有关的变量和参数，见表 16-4。

表 16-4 tDisplay

定义	typedef struct { int32_t i32Size; void *pvDisplayData; uint16_t ui16Width; uint16_t ui16Height; void (*pfnPixelDraw)(void *pvDisplayData, int32_t i32X, int32_t i32Y, uint32_t ui32Value); void (*pfnPixelDrawMultiple)(void *pvDisplayData, int32_t i32X, int32_t i32Y, int32_t i32X0, int32_t i32Count, int32_t i32BPP, const uint8_t *pui8Data, const uint8_t *pui8Palette); void (*pfnLineDrawH)(void *pvDisplayData, int32_t i32X1, int32_t i32X2, int32_t i32Y, uint32_t ui32Value); void (*pfnLineDrawV)(void *pvDisplayData, int32_t i32X, int32_t i32Y1, int32_t i32Y2, uint32_t ui32Value); void (*pfnRectFill)(void *pvDisplayData, const tRectangle *psRect, uint32_t ui32Value); uint32_t (*pfnColorTranslate)(void *pvDisplayData, uint32_t ui32Value); void (*pfnFlush)(void *pvDisplayData); } tDisplay
成　员	描　述
i32Size	结构体的大小
pvDisplayData	指向特定显示驱动数据的指针
ui16Width	显示宽度
ui16Height	显示高度
pfnPixelDraw	指向在显示设备上画点的函数指针
pfnPixelDrawMultiple	指向在显示设备上画多点的函数指针
pfnLineDrawH	指向在显示设备上绘制水平线的函数指针
pfnLineDrawV	指向在显示设备上绘制垂直线的函数指针
pfnRectFill	指向显示设备上绘制填充型矩形的函数指针
pfnColorTranslate	指向将 24 位 RGB 颜色转换成特定显示颜色函数的指针
pfnFlush	指向在显示设备上刷新任何绘图操作缓存的函数指针
功能描述	该结构体定义了显示驱动的特性

例如，在 Tiva TM4C123G 开发板中，采用 CFAL9664 - F - B1 OLED 和 SSD1332 控制芯片液晶屏的 tDisplay 结构体的定义如下：

```
const tDisplay g_sCFAL96x64x16 =
{
```

```
        sizeof(tDisplay),                    //结构体的大小
        0,                                   //没有用到特定数据,这里为空
        96,                                  //显示宽度
        64,                                  //显示高度
        CFAL96x64x16PixelDraw,               //画点
        CFAL96x64x16PixelDrawMultiple,       //画多点
        CFAL96x64x16LineDrawH,               //绘制水平线
        CFAL96x64x16LineDrawV,               //绘制垂直线
        CFAL96x64x16RectFill,                //绘制填充型矩形
        CFAL96x64x16ColorTranslate,          //将 24 位 RGB 颜色转换成特定显示颜色
        CFAL96x64x16Flush                    //刷新任何绘图操作缓存
    };
```

在这个结构体中定义了 OLED 屏的色彩数据为 16BPP，显示区大小(96(宽)×64(高))，以及调用了基本的图形绘制函数。

（3）显示驱动所使用的固件库函数介绍

① ColorTranslate 函数，见表 16-5。

表 16-5　ColorTranslate

功能	将 24 位的 RGB 颜色转换成显示驱动特定的颜色
函数原型	int32_t　ColorTranslate(void　* pvDisplayData,uint32_t ui32Value)
输入参数	描述
pvDisplayData ui32Value	指向特定显示驱动数据的指针 24 位 RGB 颜色
功能描述	该函数用于把 24 位的 RGB 颜色的值转换成可被写入到显示帧缓冲区来再现该颜色的值
返回参数	返回显示驱动程序特定的颜色

② Flush 函数，见表 16-6。

表 16-6　Flush

功能	刷新任何缓存的绘制操作
函数原型	void　Flush（void　* pvDisplayData）
输入参数	描述
pvDisplayData	指向特定显示驱动数据的指针
功能描述	这个函数刷新任何绘图操作显示缓存。这有利于本地帧缓冲区用于绘图操作，以及刷新将复制到显示器的本地帧缓冲区。如果没有缓存操作的显示驱动，此函数可以为空
返回参数	无

③ LineDrawH 函数，见表 16-7。

表 16-7　LineDrawH

功能	绘制一条水平线
函数原型	void　LineDrawH(void　* pvDisplayData, i32X1, i32X2, i32Y, Uint_32 ui32Value)

（续）

输入参数	描述
pvDisplayData i32X1 i32X2 i32Y ui32Value	指向特定显示驱动数据的指针 线的 X 起始坐标 线的 X 结束坐标 线的 Y 坐标 线的颜色
功能描述	该函数在显示器上绘制一条水平线，假设线的坐标在显示器的范围之内
返回参数	无

④ LineDrawV 函数，见表 16-8。

表 16-8　LineDrawV

功能	绘制一条垂直线
函数原型	void　LineDrawH (void ＊pvDisplayData, i32Y1, i32Y2, i32X Uint_32 ui32Value) d long ulValue)
输入参数	描述
pvDisplayData i32X1 i32X2 i32Y ui32Value	指向特定显示驱动数据的指针 线的 Y 起始坐标 线的 Y 结束坐标 线的 X 坐标 线的颜色
功能描述	该函数在显示器上绘制一条垂直线，假设线的坐标在显示器的范围之内
返回参数	无

⑤ PixelDraw 函数，见表 16-9。

表 16-9　PixelDraw

功能	在屏幕上绘制一个像素点
函数原型	void　PixelDraw(void ＊pvDisplayData, i32X, i32Y, Uint_t ui32Value)
输入参数	描述
pvDisplayData i32X i32Y ui32Value	指向特定显示驱动数据的指针 像素点的 X 坐标 像素点的 Y 坐标 像素的颜色
功能描述	该函数设置给定像素为一个特定颜色
返回参数	无

⑥ PixelDraw 函数，见表 16-10。

表 16-10　PixelDraw

功能	在屏幕上绘制一个像素点
函数原型	void　PixelDraw(void ＊pvDisplayData, i32X, i32Y, Uint_t ui32Value)
输入参数	描述
pvDisplayData i32X 32iY ui32Value	指向特定显示驱动数据的指针 像素点的 X 坐标 像素点的 Y 坐标 像素的颜色
功能描述	该函数设置给定像素为一个特定颜色
返回参数	无

⑦ PixelDrawMultiple 函数，见表 16-11。

表 16-11　PixelDrawMultiple

功能	在屏幕上绘制水平像素点
函数原型	void　PixelDraw Multiple (void ＊pvDisplayData, i32X, i32Y, i32X0, i32Count, i32BPP, Const uint8_t　＊pi8Data, Const uint8_t　＊)
输入参数	描述
pvDisplayData i32X i32Y i32X0 i32Count i32BPP pi8Data pi8Palette	指向特定显示驱动数据的指针 第一个像素点的 X 坐标 第一个像素点的 Y 坐标 子像素在像素数据内的偏移量，在每像素格式 1 或 4 位格式中的子像素偏移量是有效的 待绘制的像素点数目 每像素的位数，其必须为 1、4 或 8 指向像素数据的指针 指向用于绘制像素调色板的指针
功能描述	该函数使用提供的调色板在屏幕上绘制水平像素点
返回参数	无

⑧ RectFill 函数，见表 16-12。

表 16-12　RectFill

功能	填充矩形
函数原型	void　RectFill(void ＊pvDisplayData, const tRectangle ＊pRect, uint32_t ui32Value)
输入参数	描述
pvDisplayData pRect 32uiValue	指向特定显示驱动数据的指针 指向描述矩形结构体的指针 矩形的颜色

（续）

功能描述	该函数在显示器上填充一个矩形
返回参数	无

（4）OELD 液晶屏驱动程序片段解读

下面将对 TI 提供的 CFAL96x64 OLED 液晶屏的驱动程序片段作一比较详细的介绍。

```
// ****************************************************************************
//定义所使用的 SSI 外设和数据速率
// ****************************************************************************

#define DISPLAY_SSI_BASE            SSI2_BASE //SSI2
#define DISPLAY_SSI_CLOCK           4000000

// ****************************************************************************
//定义显示数据/命令(D/ C)信号的端口和引脚
// ****************************************************************************
#define DISPLAY_D_C_PORT            GPIO_PORTH_BASE
#define DISPLAY_D_C_PIN             GPIO_PIN_6

//当初始化时发送给显示器的命令数组
static
uint8_t g_ui8DisplayInitCommands[ ] =
{
//   0xAE,              //0xAE 是命令表中的关闭显示命令(见表 16-2)
     0x87,0x07,         //0x87 命令表中主控制电流的衰减因子 (因子 =7/16)
     0x81,0xA0,         //0x81 命令表中对比度控制 A 命令,共 256 级,这里为 160
     0x82,0x60,         //0x82 命令表中对比度控制 B 命令,共 256 级,这里为 96
     0x83,0xB0,         //0x83 命令表中对比度控制 C 命令,共 256 级,这里为 176
     0xA0,0x20,         //0xA0 重新映射和数据格式命令,“00”表示使用 8BPP 色彩模式
     0x26,0x01,         //0x26 填充使能/禁止命令,0x01 为矩形填充使能
     0xAF               //0xAF 为打开显示命令
};

// ****************************************************************************
//! 向显示控制器写入一组指令字节
//! 参数 pi8Cmd 是指向一组命令字节的指针
//! 参数 ui32Count 是命令的字节数
//! 此函数提供了一种发送多个命令字节到显示控制器的方法
//! 它可以用于单个指令或多个命令在缓冲器中链接一起
//! 它会等待任何以前的操作完成,然后把所有的命令字节复制到控制器中
//! 直到最后一个命令字节被写入到 SSI 的 FIFO 中时,它才会返回。但当该函数返回时
//! 数据仍然可以被移出到显示控制器
// ****************************************************************************
static void
CFAL96x64x16WriteCommand(const uint8_t * pi8Cmd,uint32_t ui32Count)
{
    //
```

```
    //等待以前的操作完成
    //
    while(ROM_SSIBusy(DISPLAY_SSI_BASE))
    {
    }

    //
    //将 D/C 引脚的电平拉低来指示是写命令
    //
    ROM_GPIOPinWrite(DISPLAY_D_C_PORT,DISPLAY_D_C_PIN,0);
    //
    //向显示器发送所有命令字节
    //
    while(ui32Count--)
    {
        ROM_SSIDataPut(DISPLAY_SSI_BASE, *pi8Cmd);
        pi8Cmd++;
    }
}

//*****************************************************************************
//! 向显示控制器写入一组数据字节
//*****************************************************************************
static void
CFAL96x64x16WriteData(const uint8_t *pi8Data,uint32_t ui32Count)
{
    //
    //等待以前的操作完成
    //
    while(ROM_SSIBusy(DISPLAY_SSI_BASE))
    {
    }

    //
    //将 D / C 引脚置为高电平来指示是写数据
    //
    ROM_GPIOPinWrite(DISPLAY_D_C_PORT,DISPLAY_D_C_PIN,DISPLAY_D_C_PIN);

    //
    //向显示器发送所有数据字节
    //
    while(ui32Count--)
    {
        ROM_SSIDataPut(DISPLAY_SSI_BASE, *pi8Data);
        pi8Data++;
    }
}

//*****************************************************************************
```

```
//! 在屏幕上绘制一个像素点
// ****************************************************************************
static void
CFAL96x64x16PixelDraw(void *pvDisplayData,int32_t i32X,int32_t i32Y,
                      uint32_t ui32Value)
{
    uint8_t ui8Cmd[8];

    //
    //加载列命令,开始和结束列
    //
    ui8Cmd[0] = 0x15;              //0x15 是列地址设置命令(见表 16-2)
    ui8Cmd[1] = (uint8_t)i32X;   //开始列
    ui8Cmd[2] = (uint8_t)i32X;   //结束列

    //
    //加载行命令,开始和结束行
    //
    ui8Cmd[3] = 0x75;              //0x75 是行地址设置命令(见表 16-2)
    ui8Cmd[4] = (uint8_t)i32Y;   //开始行
    ui8Cmd[5] = (uint8_t)i32Y;   //结束行

    //
    //把数组 ui8Cmd[8]的前 6 个元素送给显示,即送行、列命令给显示
    //
    CFAL96x64x16WriteCommand(ui8Cmd,6);

    //
    //把表示像素的数据值发送到显示器
    //
    CFAL96x64x16WriteData((uint8_t *)&ui32Value,1);
}

// ****************************************************************************
//! 画水平线段
// ****************************************************************************
static void
CFAL96x64x16LineDrawH(void *pvDisplayData,int32_t i32X1,int32_t i32X2,
                      int32_ t i32Y,uint32_t ui32Value)
{
    uint8_t ui8LineBuf[16];
    unsigned int uIdx;

    //
    //发送行、列的起始和结束地址
    //
    ui8LineBuf[0] = 0x15;            //0x15 是列地址设置命令
    ui8LineBuf[1] = i32X1 < i32X2 ? i32X1:i32X2;      //开始列
    ui8LineBuf[2] = 95;              //结束列
```

```
ui8LineBuf[3] =0x75;                //0x75 是行地址设置命令
    ui8LineBuf[4] = i32Y;           //开始行
    ui8LineBuf[5] =63;              //结束行

CFAL96x64x16WriteCommand(ui8LineBuf,6);   //发送 ui8LineBuf[16]数组的前 6 个元素
                                    //给写命令函数

    //
    //使用缓冲器中的像素画线,所以可在同一时间发送多个字节;用线的颜色填充缓冲区
    //
    for(uIdx =0; uIdx < sizeof(ui8LineBuf); uIdx ++)
    {
        ui8LineBuf[uIdx] = ui32Value;
    }

    uIdx = (i32X1 < i32X2) ? (i32X2 - i32X1) : (i32X1 - i32X2);
    uIdx += 1;
    while(uIdx)
    {
        CFAL96x64x16WriteData(ui8LineBuf,(uIdx < 16) ? uIdx : 16);
        uIdx  -= (uIdx < 16) ? uIdx:16;
    }
}

// *****************************************************************************
//! 初始化显示驱动程序
//! 该函数初始化面板上的 SSD1332 显示控制器,准备要显示的数据
//! 通信采用串行 SPI 方式
// *****************************************************************************
void
CFAL96x64x16Init(void)
//PinoutSet(void)                   //设置引脚及外设功能函数
{
    tRectangle sRect;

    //
    //使能该驱动程序的外设
    //
    ROM_SysCtlPeripheralEnable(DISPLAY_SSI_PERIPH);
    ROM_SysCtlPeripheralEnable(DISPLAY_SSI_GPIO_PERIPH);
    ROM_SysCtlPeripheralEnable(DISPLAY_RST_GPIO_PERIPH);

    //
    //选择 SSI 功能的相应引脚
    //
    ROM_GPIOPinConfigure(DISPLAY_PINCFG_SSICLK);
    ROM_GPIOPinConfigure(DISPLAY_PINCFG_SSIFSS);
    ROM_GPIOPinConfigure(DISPLAY_PINCFG_SSITX);
```

```
//
//配置引脚为 SSI 功能
//
ROM_GPIOPinTypeSSI(DISPLAY_SSI_PORT,DISPLAY_SSI_PINS);

//
//配置显示控制引脚为 GPIO 输出
//
ROM_GPIOPinTypeGPIOOutput(DISPLAY_RST_PORT,DISPLAY_RST_PIN);
ROM_GPIOPinTypeGPIOOutput(DISPLAY_ENV_PORT,DISPLAY_ENV_PIN);
ROM_GPIOPinTypeGPIOOutput(DISPLAY_D_C_PORT,DISPLAY_D_C_PIN);

//
//复位引脚为高电平,关闭电源
//
ROM_GPIOPinWrite(DISPLAY_RST_PORT,DISPLAY_RST_PIN,DISPLAY_RST_PIN);
ROM_GPIOPinWrite(DISPLAY_ENV_PORT,DISPLAY_ENV_PIN,0);
ROM_SysCtlDelay(1000);

//
//在执行其他程序时,驱动复位引脚可拉低
//
ROM_GPIOPinWrite(DISPLAY_RST_PORT,DISPLAY_RST_PIN,0);

//
//配置 SSI 端口
//
ROM_SSIDisable(DISPLAY_SSI_BASE);
ROM_SSIConfigSetExpClk(DISPLAY_SSI_BASE,ROM_SysCtlClockGet(),
                       SSI_FRF_MOTO_MODE_3,SSI_MODE_MASTER,
                       DISPLAY_SSI_CLOCK,8);
ROM_SSIEnable(DISPLAY_SSI_BASE);

//
//复位后显示
//
ROM_SysCtlDelay(1000);
ROM_GPIOPinWrite(DISPLAY_RST_PORT,DISPLAY_RST_PIN,DISPLAY_RST_PIN);
ROM_SysCtlDelay(1000);

//
//使能显示器电源
//
ROM_GPIOPinWrite(DISPLAY_ENV_PORT,DISPLAY_ENV_PIN,DISPLAY_ENV_PIN);
ROM_SysCtlDelay(1000);

//
//发送初始配置命令字节到显示
```

```
    //
    CFAL96x64x16WriteCommand(g_ui8DisplayInitCommands,
                             sizeof(g_ui8DisplayInitCommands));
    ROM_SysCtlDelay(1000);

    //
    //用黑色的矩形填满整个液晶屏
    //
    sRect.i16XMin = 0;          //列起始坐标
    sRect.i16XMax = 95;         //列结束坐标
    sRect.i16YMin = 0;          //行起始坐标
    sRect.i16YMax = 63;         //行结束坐标
    CFAL96x64x16RectFill(0,&sRect,0);   //第一个 0 表示:未使用特殊数据;第二个 0
                                //表示:黑色;&sRect 表示:矩形
}
```

2. 图形基元层

图形基元层提供了一组低层的绘图操作。这些操作包括画线、圆、文本和位图等。允许多个绘图操作使用离屏缓冲区，并将最后的结果一次性复制到屏幕上。

（1）常用数据结构定义

① tCodePointMap 结构体，见表 16-13。

表 16-13　tCodePointMap

定义	typedef struct { uint16_t ui16SrcCodepage; uint16_t ui16FontCodepage; uint32_t (* pfnMapChar)(const char * pcSrcChar, uint32_t ui32Count, uint32_t * pui32Skip); } tCodePointMap
成员	描述
ui16SrcCodepage ui16FontCodepage pfnMapChar	代码页用来描述源字符 映射到源字符的代码页 用于将输入字符串转换成输出代码页的代码点的转换函数指针
功能描述	用于定义一种映射函数的结构，此函数可以将代码页中的文本转换到不同的代码页。这通常用于将文本字符串转换成所需要的码点，以检索一个适当的字形字体

② tContext 结构体，见表 16-14。

表 16-14　tContext

定义	typedef struct { int32_t i32Size; const tDisplay * psDisplay; tRectangle sClipRegion; uint32_t ui32Foreground; uint32_t ui32Background; const tFont * psFont; void (* pfnStringRenderer)(const struct _tContext * ,

（续）

定义	const char *, int32_t, int32_t, int32_t, bool); const tCodePointMap * pCodePointMapTable; uint16_t ui16Codepage; uint8_t ui8NumCodePointMaps; uint8_t ui8CodePointMap; uint8_t ui8Reserved; } tContext
成员	描述
i32Size	结构体的大小
pDisplay	在屏幕上执行绘图操作
sClipRegion	在屏幕上绘图时所使用的裁减区域
ui32Foreground	在屏幕上绘制图形的前景色
ui32Background	在屏幕上绘制图形的背景色
psFont	在屏幕上渲染文本的字体
pfnStringRenderer	指向替换字符串渲染函数的指针。应用程序能够使用它为特定语言字符串提供渲染支持。如果设置了该功能，则其通过调用 GrStringDraw 函数来控制
pCodePointMapTable	用于映射在各种被支持的源代码页与使用字体所支持的代码页之间的功能表
ui16Codepage	当前选定的源文本的代码页
ui8NumCodePointMaps	在 pCodePointMapTable 数组中的条目数
ui8CodePointMap	基于选定的代码页和当前字体的代码点映射表条目的索引
ui8Reserved	保留用于将来的扩展
功能描述	该结构定义了用于在屏幕上绘图的绘图上下文，在任何时候可以存在多种绘图上下文

③ tFont 结构体，见表 16-15。

表 16-15　tFont

定义	typedef struct { uint8_t ui8Format; uint8_t ui8MaxWidth; uint8_t ui8Height; uint8_t ui8Baseline; uint16_t pui16Offset[96]; const uint8_t * pui8Data; } tFont
成员	描述
ui8Format	字体的格式，为 FONT_FMT_UNCOMPRESSED、FONT_FMT_PIXEL_RLE 其中之一
ui8MaxWidth	字符的最大宽度，这是字体中的最宽字符，任何单个的字符可能比这个宽度窄
ui8Height	字符单元的高度，这可能比字符的字体数据高（提供行间距）
ui8Baseline	字符单元的顶部和字符的字形的基线之间的偏移量。基线是大写字母的底部行，低于基线仅出现在下行小写字母的情况
pui16Offset	在字体中的每一个字符数据在 pui8Data 中的偏移量
pui8Data	指向字体数据指针
功能描述	该结构描述了在屏幕上绘制文本所用的字体，此格式的字体可以用 0x20～0x7F 范围内的代码点将字体编码成 ASCII 字符

④ tGrLibDefaults 结构体，见表 16-16。

表 16-16　tGrLibDefaults

定义	typedef struct { 　void (* pfnStringRenderer)(const struct _tContext * , 　　　　const char * , 　　　　int32_t, 　　　　int32_t, 　　　　int32_t, 　　　　bool); 　tCodePointMap * pCodePointMapTable; 　uint16_t ui16Codepage; 　uint8_t ui8NumCodePointMaps; 　uint8_t ui8Reserved; } tGrLibDefaults
成员	描述
pfnStringRenderer	使用的默认字符串渲染函数，一般为 GrDefaultStringRenderer 函数，但当支持语言需由如阿拉伯语和希伯来语混合来直接渲染时，则将被替换
pCodePointMapTable	默认的代码点映射函数表，这个表包含允许在源代码页中映射 GrLib 文本信息为使用中字体的正确字形。指向字段结构体数组的第一个元素
ui16Codepage	由应用程序使用的默认源文本代码页编码
ui8NumCodePointMaps	在 pCodePointMapTable 数组中的条目数量
ui8Reserved	保留
功能描述	该结构包含在通过调用 GrContextInit 初始化任何新上下文设置的默认值，该结构通过使用 GrLibInit 函数传递到图形库

⑤ tRectangle 结构体，见表 16-17。

表 16-17　tRectangle

定义	Typedef struct { 　int16_t i16XMin; 　int16_t i16YMin; 　int16_t i16XMax; 　int16_t i16YMax; } tRectangle
成员	描述
i16XMin i16YMin i16XMax i16YMax	矩形的最小 X 坐标 矩形的最小 Y 坐标 矩形的最大 X 坐标 矩形的最大 Y 坐标
功能描述	该结构体定义了矩形的坐标信息，所有点坐标等于或大于最小坐标，小于或等于最大坐标

（2）常用图形基元层的固件库函数

常用图形基元层的固件库函数见表 16-18 ~ 16-31。

表 16-18　GrCircleDraw

功能	画一个圆
函数原型	void GrCircleDraw(const tContext * pContext, 　　int32_t i32X, 　　int32_t i32Y, 　　int32_t i32Radius)

（续）

参数	描述
pContext i32X i32Y i32Radius	指向使用的绘制上下文的指针 圆心的 X 坐标 圆心的 Y 坐标 圆的半径
功能描述	此函数利用 Bresenham 绘图算法绘制一个圆。圆的范围：x 坐标从 i32X - i32Radius 到 i32X + i32Radius；y 坐标从 i32Y - i32Radius 到 i32Y + i32Radius（包括边界）
返回参数	无

表 16-19　GrCircleFill

功能	画一个实心圆
函数原型	void GrCircleFill(const tContext * pContext, int32_t i32X, int32_t i32Y, int32_t i32Radius)
参数	描述
pContext i32X i32Y i32Radius	指向使用的绘制上下文的指针 圆心的 X 坐标 圆心的 Y 坐标 圆的半径
功能描述	此函数利用 Bresenham 绘图算法绘制一个实心的圆。圆的范围：x 坐标从 i32X - i32Radius 到 i32X + i32Radius；y 坐标从 i32Y - i32Radius 到 i32Y + i32Radius（包括边界）
返回参数	无

表 16-20　GrContextFontSet

功能	设置在使用中的字体
函数原型	void GrContextFontSet(tContext * pContext, const tFont * psFont)
参数	描述
pContext pFont	指向修改的绘图上下文的指针 指向被使用字体的指针
功能描述	该函数用于在指定的绘图上下文中设置字体，以便进行符串的绘图操作
返回参数	无

表 16-21　GrContextInit

功能	初始化绘图上下文
函数原型	void GrContextInit(tContext * psContext, const tDisplay * psDisplay)
输入参数	描述
pContext pDisplay	指向待初始化的绘图上下文的指针 指向描述使用显示驱动的 tDisplayInfo 结构的指针
功能描述	该函数初始化一个绘图上下文来准备使用。所提供的显示驱动程序将被用于所有后续的图形操作，且默认的裁剪区域将被设置成屏幕的范围
返回参数	无

表 16-22　GrFontHeightGet

功能	获取字体的高度
函数原型	uint32_t　GrFontHeightGet（const tFont *psFont）
输入参数	描述
pFont	指向待查询字体的指针
功能描述	该函数决定一个字体的高度。字体高度为字体的顶部和底部的字体之间的偏移量，包括任何上半格和下半格的字母
返回参数	返回以像素为单位的字体高度

表 16-23　GrFontMaxWidthGet

功能	获取字体最大宽度
函数原型	uint32_t　GrFontMaxWidthGet（const tFont *psFont）
输入参数	描述
pFont	指向待查询字体的指针
功能描述	此函数用于确定一个字体的最大宽度。最大宽度为字体中单个字符最宽的宽度
返回参数	返回以像素为单位的字体最大宽度

表 16-24　GrFontNumBlocksGet

功能	返回字体中字符编码块的数量
函数原型	uint16_t GrFontNumBlocksGet（const tFont *psFont）
输入参数	描述
pFont	指向要字体查询的指针
功能描述	此函数可用于查询给定字体编码代码点（字符）连续块的数量。这主要用于直接解析字体，例如，显示字体中的所有字形
返回参数	返回字体中代码点块的数目

表 16-25　GrImageDraw

功能	绘制位图图像
函数原型	void　GrImageDraw(const tContext *pContext, const uint8_t *pui8Image, int32_t i32X, int32_t i32Y)
输入参数	描述
pContext pui8Image i32X i32Y	指向使用的绘图上下文的指针 指向待绘制的图像的指针 图像左上角的 X 坐标 图像左上角的 Y 坐标
功能描述	此函数用于绘制位图图像。图像可以是每像素 1 位（用于绘图上下文中使用前景色和背景色），每像素 4 位（为调色板中提供图像数据），或每像素 8 位（为调色板中提供图像数据）图像数据可以是未压缩的数据，也可以使用 Lempel - Ziv - Storer - Szymanski 算法对其进行压缩
返回参数	无

表 16-26　GrLineDraw

功能	绘制一条直线
函数原型	void GrLineDraw(const tContext * pContext, int32_t i32X1, int32_t i32Y1, int32_t i32X2, int32_t i32Y2)
输入参数	描述
pContext i32X1 i32Y1 i32X2 i32Y2	指向要绘图上下文的指针 直线起始位置的 X 坐标 直线起始位置的 Y 坐标 直线结束位置的 X 坐标 直线结束位置的 Y 坐标
返回参数	无

表 16-27　GrLineDrawH

功能	绘制一条水平线
函数原型	void GrLineDrawH(const tContext * pContext, int32_t i32X1, int32_t i32X2, int32_t i32Y)
输入参数	描述
pContext i32X1 i32X2 i32Y	指向要绘图上下文的指针 直线起始位置的 X 坐标 直线结束位置的 X 坐标 直线的 Y 坐标
返回参数	无

表 16-28　GrLineDrawV

功能	绘制一条垂直线
函数原型	void GrLineDrawV(const tContext * pContext, int32_t i32X, int32_t i32Y1, int32_t i32Y2)
输入参数	描述
pContext i32X i32Y1 i32Y2	指向要绘图上下文的指针 直线的 X 坐标 直线起始位置的 Y 坐标 直线结束位置的 Y 坐标
返回参数	无

表 16-29　GrRectDraw

功能	绘制一个矩形
函数原型	void　GrRectDraw(const tContext * pContext,const tRectangle * pRect)
输入参数	描述

（续）

pContext pRect	指向待绘图上下文的指针 指向包含矩形内容的结构指针
返回参数	无

表 16-30　GrRectFill

功能	绘制一个填充的矩形
函数原型	void GrRectFill(const tContext ＊pContext,const tRectangle ＊pRect)
输入参数	描述
pContext pRect	指向待绘图上下文的指针 指向包含矩形内容的结构指针
返回参数	无

表 16-31　GrStringDraw

功能	绘制一个字符串
函数原型	void GrStringDraw(const tContext ＊pContext, const char ＊pcString, int32_t i32Length, int32_t i32X, int32_t i32Y, uint32_t bOpaque)
输入参数	描述
pContext pcStringi32Length i32X i32Y bOpaque	指向要绘图上下文的指针 指向要绘制的字符串指针 应绘制在屏幕上的字符串的字符数 屏幕上字符串的起始位置 X 坐标 屏幕上字符串的起始位置 Y 坐标 为真时，可以绘制每一个字符的背景色，为假时则不可以
返回参数	无

3. 小工具层

小工具层位于图形基元层之上，由如下小工具组成：

① 画布小工具（Canvas Widget）。

② 复选框小工具（Checkbox Widget）。

③ 容器小工具（Container Widget）。

④ 图形按钮小工具（Image Button Widget）。

⑤ 列表框小工具（ListBox Widget）。

⑥ 键盘小工具（Keyboard Widget）。

⑦ 按钮小工具（Push Button Widget）。

⑧ 单选按钮小工具（Radio Button Widget）。

⑨ 滑块小工具（Slider Widget）。

在遇到绘制以上列出的小工具图形时，仅需简单调用这些函数即可实现，比前面的显示驱动层和图像基元层绘图来得方便。这些固件库图形函数请参考 TI 的图形库函数（SW -

TM4C－GRL－UG－2.0.1.11577）的小工具部分。

16.2.4 实用工具（Utilities）

在 TivaWare 软件包的 tools/bin 目录下，提供了几个处理与 TivaWare 图形库不兼容的文字、图形、制作字符串表的工具：ftrasterize、pnmtoc、mkstringtable，以及在 tools/lmi－button 目录下设计按键的 lmi－button 脚本。下面仅就 ftrasterize 和 pnmtoc 工具的使用方法作一简介。

1. ftrasterize

可用 ftrasterize 工具来生成与 TivaWare 图形库兼容的字体，其操作步骤如下：

1）创建一个转换字体的目录。为了防止 Windows 操作系统某个字体由于操作字体转换而被无意中删除，最好在 C:\创建一个目录，例如：C:\Makemyfontandpicture。由于 ftrasterize 工具支持 TrueType®、OpenType®、PostScript® Type 1 Windows® FNT 字体的转换，因此可在 C:\Windows\Fonts 目录下把 TrueType 字体文件复制到所创建的目录下，比如复制 Arial 字体。然后再把 ftrasterize 工具也复制到 C:\Makemyfontandpicture 中，如图 16-23 所示。



图 16-23　选中的 TrueType 字体文件与 ftrasterize 工具

2）字体转换。

① ftrasterize 的命令格式：

```
ftrasterize [ -a <num> ] [ -b] [ -c <filename> ] [ -d] [ -e <num> ] [ -f <name> ]
        [ -h] [ -i] [ -m] [ -n] [ -o <num> ] [ -p <num> ] [ -r] [ -s [F] <size> ]
        [ -t <num> ] [ -u] [ -v] [ -w <num> ] [ -y] [ -z <num> ] <font>
```

其中常用的几个参数含义为：－b 为黑体；－f <name> 为这个字体的名称；－s <size> 指定字体大小；－m 指定输出的字体要等宽。其他参数介绍请参考图形库的 Utilities 部分。

② 转换操作。在命令行中输入：ftrasterize　－f　arial　－s　30　arial.ttf，如图 16-24 所示。

2. pnmtoc

可用 pnmtoc 工具来生成与 TivaWare 图形库兼容的图片，操作步骤如下：

1）下载并安装 netpbm（pnm）格式的图片转换工具 GIMP。在 http://www.gimp.org 下载 jpeg、GIF、BMP、TIFF 等格式图片转换成 netpbm（pnm）格式的工具 GIMP，选择用户自定义安装。在安装时选中如图 16-25 所示的选项。

2）将 jpeg 等格式图片转换成 pnm 格式图片，如图 16-26 与 16-27 所示。



图 16-24　字体转换操作

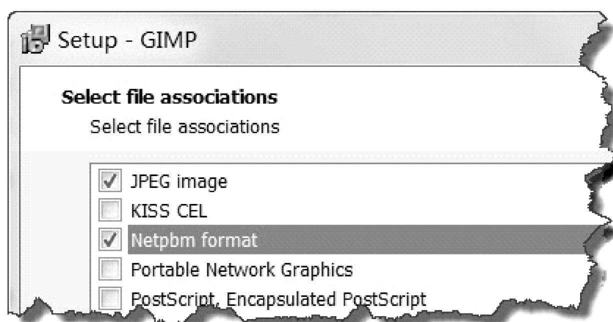


图 16-25　安装时需选中的选项





图 16-26　jpeg 格式→pnm 格式图片的转换过程

图 16-27　选中 256 色

3）保存转换后的图片，将扩展名修改为 . pnm，如图 16-28 所示。

4）用 pnmtoc 工具生成 . c 格式的图片数据。在命令行中输入：pnmtoc – c GIMP. pnm > GIMP. c，如图 16-29 所示。



图 16-28　将 GIMP. XCF 修改为 GIMP. pnm

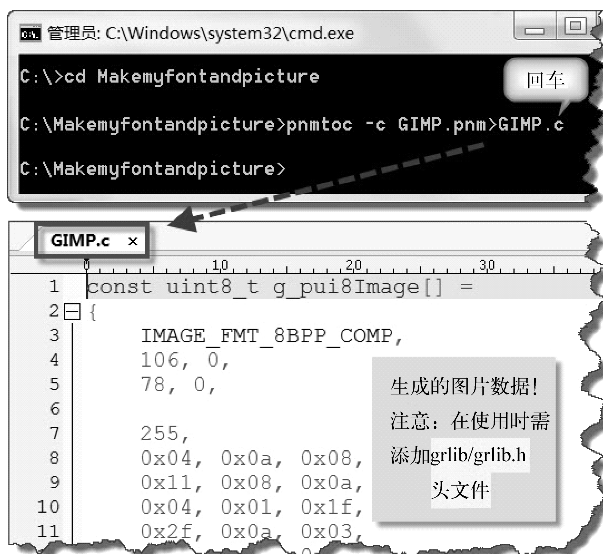


图 16-29　用 pnmtoc 工具产生的图片数据

16.2.5　预定义的颜色参考

在 grlib. h 中预定义了一组用于在屏幕上绘制图形的颜色，而其他颜色则可通过指定 RGB 的值来得到。预定义颜色给基于图形的应用带来了很大方便，其样式如图 16-30 所示。所有预定于的颜色请参考图形库中的“15 Predefined Color Reference”（269～273 页）部分。



图 16-30　预定于颜色的样式



16.3 例程

在前几章的例程中已使用过用 TFT－LCD 液晶屏来显示 TM4F 处理器输出的信息，为了便于更多的读者对图像库例程在 BD－LM4F232－UM（相似板卡 dk－tm4c123g）进行验证，下面将以 TI 提供的 grlib_demo 例程为例来介绍图形库的使用及编程方法。

1）grlib_demo. c 介绍。

```
//*****************************************************************************
//文件名:grlib_demo. c - Demonstration of the TivaWare Graphics Library.
//来源:TI 例程
//功能:该例程演示了基本图形的绘制方法,包括:画线、画圆、画矩形、显示字符串和图
//形等
//*****************************************************************************

#include <stdint. h>
#include <stdbool. h>
#include "inc/hw_sysctl. h"
#include "driverlib/rom. h"
#include "driverlib/sysctl. h"
#include "grlib/grlib. h"
#include "drivers/cfal96x64x16. h"

//*****************************************************************************
//图形上下文用于文字在 CSTN 屏上的显示
//*****************************************************************************
tContext g_sContext;

//*****************************************************************************
//TI 标志图形数据
//*****************************************************************************
const uint8_t g_pui8Logo[] =
{
    IMAGE_FMT_4BPP_COMP,
    30,0,
    30,0,
    …                    //参考 dk - tm4c123g 板中的 grlib_demo 例程相关部分
    0x70,0x40,0x17,0x47,0x77
};

//*****************************************************************************
//如果驱动程序库遇到错误将调用错误处理程序
//*****************************************************************************
#ifdef DEBUG
void
__error__(char *pcFilename,uint32_t ui32Line)
{
}
```

```
#endif

// ******************************************************************************
//一个简单的 TivaWare 图形库特性的例程
// ******************************************************************************
int
main(void)
{
    uint32_t ui32Idx;
    tRectangle sRect;

    //
    //中断处理程序使能扩展堆栈
    //
    ROM_FPULazyStackingEnable();

    //
    //设置从 PLL 运行的时钟为 50MHz
    //
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_XTAL_16MHZ |
                       SYSCTL_OSC_MAIN);

    //
    //初始化显示驱动程序
    //
    CFAL96x64x16Init();

    //
    //初始化图形上下文
    //
    GrContextInit(&g_sContext,&g_sCFAL96x64x16);

    //
    //用蓝色填充屏幕顶部的 12 行以创建一个蓝色条
    //
    sRect.i16XMin = 0;
    sRect.i16YMin = 0;
    sRect.i16XMax = GrContextDpyWidthGet(&g_sContext) - 1;
    sRect.i16YMax = 11;
    GrContextForegroundSet(&g_sContext,ClrDarkBlue);            //前景:深蓝
    GrRectFill(&g_sContext,&sRect);                             //填充矩形

    //
    //在蓝色条的周围套一个白框
    //
    GrContextForegroundSet(&g_sContext,ClrWhite);               //前景:白色

    GrRectDraw(&g_sContext,&sRect);
```

```
//
//把程序名放在蓝色条的中间
//
GrContextFontSet(&g_sContext,g_psFontFixed6x8);
GrStringDrawCentered(&g_sContext,"grlib_demo",-1,
                     GrContextDpyWidthGet(&g_sContext)/2,5,0);

//
//画一条垂直扫描线由红→绿
//
for(ui32Idx=0;ui32Idx<=8; ui32Idx++)
{
    GrContextForegroundSet(&g_sContext,
                           (((((10-ui32Idx) * 255)/8) << ClrRedShift) |
                            (((ui32Idx * 255)/8) << ClrGreenShift)));
    GrLineDraw(&g_sContext,60,60,0,60-(5 * ui32Idx));
}

//
//画一条水平扫描线由红绿→蓝
//
for(ui32Idx=1; ui32Idx <= 11; ui32Idx++)
{
    GrContextForegroundSet(&g_sContext,
                           (((((11-ui32Idx) * 255)/11) <<
                             ClrGreenShift) |
                            (((ui32Idx * 255)/11) << ClrBlueShift)));
    GrLineDraw(&g_sContext,60,60,(ui32Idx * 5),20);
}

//
//绘制重叠的实心圆
//
GrContextForegroundSet(&g_sContext,ClrBlue);                //前景:蓝色
GrCircleFill(&g_sContext,80,30,15);                         //用蓝色填充圆
GrContextForegroundSet(&g_sContext,ClrWhite);               //前景:白色
GrCircleDraw(&g_sContext,80,30,15);                         //绘制圆

//
//绘制重叠的矩形
//
GrContextForegroundSet(&g_sContext,ClrGray);                //前景:灰色
sRect.i16XMin=8;
sRect.i16YMin=45;
sRect.i16XMax=46;
sRect.i16YMax=51;
GrRectFill(&g_sContext,&sRect);                             //用灰色填充矩形
GrContextForegroundSet(&g_sContext,ClrWhite);               //前景:白色
sRect.i16XMin+=4;
```

```
        sRect.i16YMin += 4;
        sRect.i16XMax += 4;
        sRect.i16YMax += 4;
        GrRectDraw(&g_sContext,&sRect);                         //画一个矩形

        //画一段增大尺寸字体的文本
        //Draw a piece of text in fonts of increasing size.
        //
        GrContextForegroundSet(&g_sContext,ClrBlack);
        GrStringDraw(&g_sContext,"Strings", -1,6,16,0);
        GrContextForegroundSet(&g_sContext,ClrSilver);
        GrStringDraw(&g_sContext,"Strings", -1,7,17,0);

        //
        //绘制图像
        //
        GrTransparentImageDraw(&g_sContext,g_pui8Logo,64,34,ClrBlack);
    #if 0
        GrImageDraw(&g_sContext,g_pui8Logo,64,34);
    #endif

        //
        //刷新所有绘制操作的缓冲区
        //
        GrFlush(&g_sContext);

        //
        //无限循环
        //
        while(1)
        {
        }
    }
```

2）在 Keil4.73 中导入 C:\ti\TivaWare_C_Series-2.0.1.11577\examples\boards\dk-tm4c123g 目录下的 grlib_demo 工程，如图 16-31 所示。

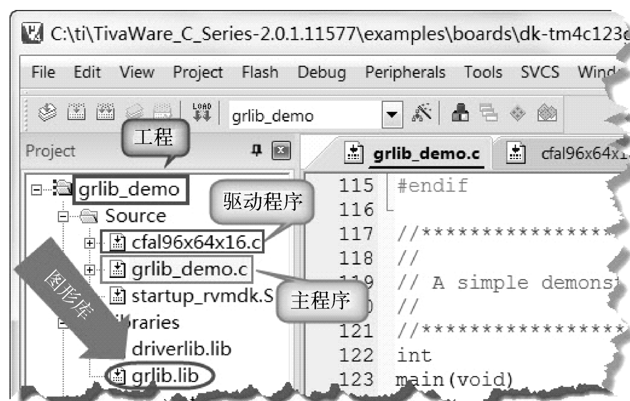


图 16-31　导入 grlib_demo 工程

注意：如果读者在安装 CCS6. x、Keil4. x 或 IAR6. x 软件时，没有一并安装图形库软件（grlib. lib），在编译 grlib_demo 工程前需先行安装该软件。

3）编译 grlib_demo 工程，并将其结果导入到 BD－LM4F232－UM 板中，其测试结果如图 16-32 所示。

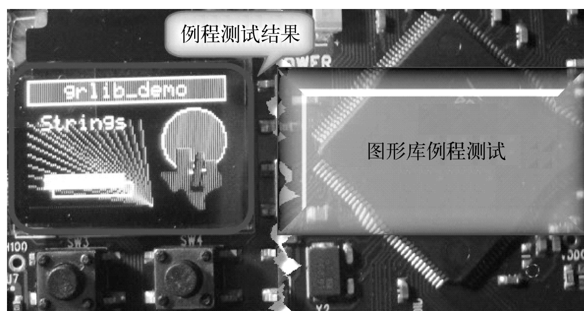


图 16-32　图形库例程在 BD－LM4F232－UM 板中的测试结果

注意：TI 还提供了一些有关图形库应用的例程，请参考 TI 网站或本书附带的资料。

附　　录



附录 A　第 3 章附录：UART 固件库函数简介

本章附录简要介绍 UART 模块的固件库函数（API）的参数定义及功能说明，详细的功能说明请参考 TI 文档：SW－TM4C－DRL－UG－2.0.1.11577。

注：为了压缩固件库功能说明的篇幅，省略掉了注意事项部分，请读者注意。

表 A-1　UART9BitAddrSend

功能	在 9 位模式下，从指定端口发送一个地址字符
函数原型	void UART9BitAddrSend（uint32_t ui32Base, uint8_t　ui8Addr）
参数	ui32Base　UART 端口的基地址 ui8Addr　待被发送的地址
描述	该函数直到所有数据都被从指定端口发送完成之后，才发送一个给定的地址作为地址字节。在返回前一直等待地址字节发送完成 常用数据函数（UARTCharPut( )、UARTCharPutNonBlocking( )、UARTCharGet( )和 UARTCharGetNonBlocking( )）可用于在 9 位模式下发送和接收数据字符
返回参数	无

表 A-2　UART9BitAddrSet

功能	设置 9 位模式的设备地址
函数原型	void UART9BitAddrSet（uint32_t ui32Base, uint8_t ui8Addr, uint8_t ui8Mask）
参数	ui32Base　UART 端口的基地址 ui8Addr　设备地址 uiMask　设备地址屏蔽
描述	该函数配置设备地址或响应 9 位 UART 端口请求的设备地址范围。接收到的地址由地址屏蔽所屏蔽，然后与给定地址比较，允许匹配成单一地址（如 uiMask 为 0xff）或一段地址
返回参数	无

表 A-3　UART9BitDisable

功能	禁止指定 UART 的 9 位模式
函数原型	void UART9BitDisable（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数禁止 UART 的 9 位操作模式
返回参数	无

表 A-4　UART9BitEnable

功能	使能指定的 UART 9 位模式
函数原型	void UART9BitEnable（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数使能 UART 的 9 位的操作模式
返回参数	无

表 A-5　UARTBreakCtl

功能	发送一个 BREAK
函数原型	void UARTBreakCtl（uint32_t ui32Base, boolbBreakState）
参数	ui32Base　UART 端口的基地址 bBreakState　输出电平控制
描述	当参数 bBreakState 设置为 true 时，调用该函数将声明一个 UART 中止条件；当 bBreak-State 设置为 false 时，调用该函数将删除中止条件。而正确的中止命令传输，必须保证至少两个完整的帧
返回参数	无

表 A-6　UARTBusy

功能	确认 UART 是否发送忙
函数原型	tBoolean UARTBusy（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数允许调用者确定是否所有的传输字节已清除了发送器硬件。如果返回 false，则发送 FIFO 为空并且最后一个传输字符的所有位（包括所有的停止位）都移出了硬件移位寄存器
返回参数	若 UART 正在发送，则返回 true；若所有传输已完成，则返回 falses

表 A-7　UARTCharGet

功能	等待指定端口的一个字符
函数原型	uint32_t UARTCharGet（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数从指定端口的接收 FIFO 中获取一个字符。如果没有可用的字符，该函数将一直等待，直到接收到一个字符方可返回
返回参数	返回从指定端口读取的字符并转换为 int32_t 类型

表 A-8　UARTCharGetNonBlocking

功能	从指定的端口接收一个字符
函数原型	uint32_t UARTCharGetNonBlocking（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数从指定端口的接收 FIFO 中获取一个字符
返回参数	返回从指定端口读取的字符并转换为 uint32_t 类型。若在当前接收 FIFO 中没有字符，则返回 -1。在尝试调用这个函数前，应该先行调用函数 UARTCharsAvail( )

表 A-9　UARTCharPut

功能	等待从指定的端口发送一个字符
函数原型	void UARTCharPut（uint32_t ui32Base, unsigned char ucData）
参数	ui32Base　UART 端口的基地址 ucData　待发送的字符
描述	该函数把字符 ucData 发送到指定端口的发送 FIFO 中。如果发送 FIFO 中没有可用空间，该函数将一直等待，直到有可用空间再返回
返回参数	无

表 A-10　UARTCharPutNonBlocking

功能	发送字符到指定端口
函数原型	bool UARTCharPutNonBlocking（uint32_t ui32Base, unsigned char ucData）
参数	ui32Base　UART 端口的基地址 ucData　待发送的字符
描述	该函数将字符 ucData 写入到指定端口的发送 FIFO 中。此函数不会阻塞（block），因此，如果没有可用的空间将返回 false，且应用程序必须在稍后重试该函数
返回参数	如果字符被成功加载到发送 FIFO 中，则返回 true；若在发送 FIFO 中没有可用的空间，则返回 false

表 A-11　UARTCharsAvail

功能	确定接收 FIFO 中是否有字符存在
函数原型	Bool UARTCharsAvail（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址 ucData　待发送的字符
描述	该函数返回一个指示接收 FIFO 中是否有数据的标志
返回参数	如果在接收 FIFO 中有数据则返回 true；若在接收 FIFO 中没有数据，则返回 false

表 A-12　UARTClockSourceGet

功能	获取指定 UART 波特率的时钟源
函数原型	uint32_t UARTClockSourceGet（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数返回指定 UART 波特率时钟源。可能的波特率时钟源为系统时钟（UART_CLOCK_SYSTEM）或内部精密振荡器（UART_CLOCK_PIOSC）
返回参数	无

表 A-13　UARTClockSourceSet

功能	设置指定 UART 波特率的时钟源
函数原型	void UARTClockSourceSet（uint32_t ui32Base, uint32_t ui32Source）

参数	ui32Base　UART 端口的基地址 ui32Source　UART 波特率时钟源
描述	该函数允许选择 UART 波特率的时钟源。可能的波特率时钟源为系统时钟（UART_CLOCK_SYSTEM）或内部精密振荡器（UART_CLOCK_PIOSC） 更改波特率的时钟源需重新配置波特率
返回参数	无

表 A-14　UARTConfigGetExpClk

功能	获取 UART 的当前配置
函数原型	void UARTConfigGetExpClk（uint32_ t ui32Base, uint32_t　ui32UARTClk, uint32_t　* pui32Baud, uint32_t　* pui32Config）
参数	ui32Base　UART 端口的基地址 ui32UARTClk　提供 UART 模块的时钟速率 pui32Baud　指向波特率存放单元的指针 pui32Config　指向数据格式存放单元的指针
描述	该函数确定 UART 的波特率和数据格式，提供一个显式的外设时钟（以 ExpClk 为后缀）。返回实际的波特率；它可能不是所需的精确波特率或“正式的”波特率。pui32Config 返回的数据格式与 UARTConfigSetExpClk( ) 的参数 ui32Config 所枚举的相同 外设时钟 = 处理器时钟。系统时钟的频率是 SysCtlClockGet( ) 的返回值，或者它为已知常量时，其可为显式的硬编码 对于 Tiva 器件，可通过 UARTClockSourceSet( ) 指定 UART 波特率的时钟源，外设时钟可以改变 PIOSC。在这种情况下，外设时钟应指定为 16 000 000
返回参数	无

表 A-15　UARTConfigSetExpClk

功能	设置 UART 的配置
函数原型	void UARTConfigSetExpClk（uint32_t ui32Base, uint32_t ui32UARTClk, uint32_t ui32Baud, uint32_t ui32Config）
参数	ui32Base　UART 端口的基地址 ui32UARTClk　提供给 UART 模块的时钟速率 ui32Baud　所需的波特率 ui32Config　端口（数据位数、停止位和奇偶校验）的数据格式
描述	该函数配置 UART，使其运行在指定的数据格式下。ui32Baud 参数提供波特率，ui32Config 参数配置数据格式。且 ui32Config 是数据位、停止位、奇偶校验位这三个值的逻辑或。 ➢ 数据位： ◇ UART_CONFIG_WLEN_8　　//8 位数据 ◇ UART_CONFIG_WLEN_7　　//7 位数据 ◇ UART_CONFIG_WLEN_6　　//6 位数据 ◇ UART_CONFIG_WLEN_5　　//5 位数据

(续)

描述	➢ 停止位： ◇ UART_CONFIG_STOP_ONE　　//1 个停止位 ◇ UART_CONFIG_STOP_TWO　　//2 个停止位 ➢ 奇偶校验位 ◇ UART_CONFIG_PAR_NONE　　//无奇偶校验位 ◇ UART_CONFIG_PAR_EVEN　　//偶校验位 ◇ UART_CONFIG_PAR_ODD　　//奇校验模 ◇ UART_CONFIG_PAR_ONE　　//1 个奇偶校验位 ◇ UART_CONFIG_PAR_ZERO　　//校验位始终为 0
返回参数	无

表 A-16　UARTDisable

功能	禁止发送和接收
函数原型	void UARTDisable（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数禁止 UART，等待当前字符发送结束，并刷新发送 FIFO
返回参数	无

表 A-17　UARTDisableSIR

功能	禁止指定 UART 的 SIR（IrDA）模式
函数原型	void UARTDisableSIR（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	此函数禁止 UART 的 SIR（IrDA）模式，清除 SIREN（IRDA）和 SIRLP 位（低功率） 该函数只在 UART 由 UARTEnable()函数使能前有效，因此在调用 UARTEnableSIR()之前，必须先行调用 UARTConfigSetExpClk()，因为 UARTConfigSetExpClk()函数需调用 UARTEnable()函数 另一种选择是先调用 UARTDisable()，其次调用 UARTEnableSIR()，然后通过调用 UARTEnable()使能 UART
返回参数	无

表 A-18　UARTDMADisable

功能	禁止 UART DMA 操作
函数原型	void UARTDMADisable（uint32_t ui32Base, uint32_t ui32DMAFlags）
参数	ui32Base　UART 端口的基地址 ui32DMAFlags　禁止 DMA 功能的位屏蔽
描述	该函数用于禁止 UARTDMAEnable（）使能的 UART DMA 功能。禁止指定的 UART DMA 功能。而参数 ui32DMAFlags 为下列任意值的逻辑或： ➢ UART_DMA_RX　　//禁止 DMA 接收 ➢ UART_DMA_TX　　//禁止 DMA 传输 ➢ UART_DMA_ERR_RXSTOP//当 UART 发生错误时不禁止 DMA 接收
返回参数	无

表 A-19　UARTDMAEnable

功能	使能 UART DMA 操作
函数原型	Void UARTDMAEnable（uint32_t ui32Base, uint32_t ui32DMAFlags）
参数	ui32Base　UART 端口的基地址 ui32DMAFlags　禁止 DMA 功能的位屏蔽
描述	使能指定的 UART DMA 功能。UART 可配置成使用 DMA 发送或接收，以及如果发生错误时禁止接收。而参数 ui32DMAFlags 为下列任意值的逻辑或： ➢ UART_DMA_RX　//使能 DMA 接收 ➢ UART_DMA_TX　//使能 DMA 传输 ➢ UART_DMA_ERR_RXSTOP //当 UART 发生错误时不禁止 DMA 接收
返回参数	无

表 A-20　UARTEnable

功能	使能发送和接收
函数原型	void UARTEnable（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数使能 UART，以及其发送和接收 FIFO
返回参数	无

表 A-21　UARTEnableSIR

功能	使能指定的 UART 的 SIR（IrDA）模式
函数原型	void UARTEnableSIR（uint32_t ui32Base, bool bLowPower）
参数	ui32Base　UART 端口的基地址 bLowPower　指示 SIR 的低功耗模式是否被使用
描述	该函数使能 SIR（IrDA）模式的 SIREN 控制位。如果 bLowPower 标志置位，则打开 SIR 低功耗模式（SIRLP 位会被置位） 该函数只在 UART 由 UARTEnable( )函数使能前有效，因此在调用 UARTEnableSIR( )之前，必须先行调用 UARTConfigSetExpClk( )，因为 UARTConfigSetExpClk( )函数需调用 UARTEnable( )函数 另一种选择是先调用 UARTDisable( )，其次调用 UARTEnableSIR( )，然后通过调用 UARTEnable( )使能 UART
返回参数	无

表 A-22　UARTFIFODisable

功能	禁止发送和接收 FIFO
函数原型	void UARTFIFODisable（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数禁止 UART 中的发送和接收 FIFO
返回参数	无

表 A-23　UARTFIFOEnable

功能	使能发送和接收 FIFO
函数原型	void UARTFIFOEnable（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数使能 UART 中的发送和接收 FIFO
返回参数	无

表 A-24　UARTFIFOLevelGet

功能	获取产生中断时的 FIFO 深度
函数原型	void UARTFIFOLevelGet（uint32_t ui32Base, uint32_t ＊pui32TxLevel, uint32_t ＊pui32RxLevel）
参数	ui32Base　UART 端口的基地址 pui32TxLevel　指向发送 FIFO 存储单元深度的指针，返回下列值之一： ➢ UART_FIFO_TX1_8　　//在 1/8 深度时产生发送中断 ➢ UART_FIFO_TX2_8　　//在 1/4 深度时产生发送中断 ➢ UART_FIFO_TX4_8　　//在 1/2 深度时产生发送中断 ➢ UART_FIFO_TX6_8　　//在 3/4 深度时产生发送中断 ➢ UART_FIFO_TX7_8　　//在 7/8 深度时产生发送中断 pui32RxLevel　指向接收 FIFO 深度存储单元的指针，返回下列值之一： ➢ UART_FIFO_RX1_8　　//在 1/8 深度时产生接收中断 ➢ UART_FIFO_RX2_8　　//在 1/4 深度时产生接收中断 ➢ UART_FIFO_RX4_8　　//在 1/2 深度时产生接收中断 ➢ UART_FIFO_RX6_8　　//在 3/4 深度时产生接收中断 ➢ UART_FIFO_RX7_8　　//在 7/8 深度时产生接收中断
描述	这个函数获取在产生发中断时送和接收的 FIFO 深度
返回参数	无

表 A-25　UARTFIFOLevelSet

功能	设置产生中断时的 FIFO 深度
函数原型	void UARTFIFOLevelSet（uint32_t ui32Base, uint32_t ui32TxLevel, uint32_t ui32RxLevel）
参数	ui32Base　UART 端口的基地址 ui32TxLevel　发送中断的 FIFO 深度，其值为如下之一： ➢ UART_FIFO_TX1_8　　//在 1/8 深度时产生发送中断 ➢ UART_FIFO_TX2_8　　//在 1/4 深度时产生发送中断 ➢ UART_FIFO_TX4_8　　//在 1/2 深度时产生发送中断 ➢ UART_FIFO_TX6_8　　//在 3/4 深度时产生发送中断 ➢ UART_FIFO_TX7_8　　//在 7/8 深度时产生发送中断 ui32RxLevel　接收中断的 FIFO 深度，取值为下列之一： ➢ UART_FIFO_RX1_8　　//在 1/8 深度时产生接收中断 ➢ UART_FIFO_RX2_8　　//在 1/4 深度时产生接收中断 ➢ UART_FIFO_RX4_8　　//在 1/2 深度时产生接收中断 ➢ UART_FIFO_RX6_8　　//在 3/4 深度时产生接收中断 ➢ UART_FIFO_RX7_8　　//在 7/8 深度时产生接收中断
描述	该函数用于配置在产生中断时发送和接收的 FIFO 深度
返回参数	无

表 A-26　UARTFlowControlGet

功能	返回正在使用的 UART 硬件流控制模式
函数原型	uint32_t UARTFlowControlGet（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数用于返回当前的硬件流控制模式
返回参数	返回当前使用的流控制模式。返回值是 UART_FLOWCONTROL_TX（若硬件发送流 CTS 被使能）、UART_FLOWCONTROL_RX（若硬件接收流 RTS 被使能）的逻辑或

表 A-27　UARTFlowControlSet

功能	设置使用的 UART 硬件流量控制模式
函数原型	void UARTFlowControlSet（uint32_t ui32Base, uint32_t ui32Mode）
参数	ui32Base　UART 端口的基地址 ui32Mode　使用的流控制模式。这个参数为 UART_FLOWCONTROL_TX 与 UART_FLOWCONTROL_RX 的逻辑或，如果禁止硬件流，则该参数为 UART_FLOWCONTROL_NONE（禁止硬件流控制）
描述	该函数配置所需的硬件流控制模式。如果 ui32Mode 中包含 UART_FLOWCONTROL_TX 标志，如果输入的 CTS 信号有效，则只可发送数据 若 ui32Mode 包含 UART_FLOWCONTROL_RX 标志，只有当存在接收 FIFO 的可用空间时，RTS 输出才由硬件控制。如果没有所需的硬件流量控制，则应传递 UART_FLOWCONTROL_NONE
返回参数	无

表 A-28　UARTIntClear

功能	清除 UART 中断来源
函数原型	void UARTIntClear（uint32_t ui32Base, uint32_t ui32IntFlags）
参数	ui32Base　UART 端口的基地址 ui32IntFlags　被清除中断源的位屏蔽
描述	清除指定 UART 的中断源，使其不再有效。该函数必须在中断处理程序调用，以防止在退出时中断再次被触发
返回参数	无

表 A-29　UARTIntDisable

功能	禁止单个 UART 中断源
函数原型	Void UARTIntDisable（uint32_t ui32Base, uint32_t ui32IntFlags）
参数	ui32Base　UART 端口的基地址 ui32IntFlags　禁止中断源的位屏蔽
描述	该函数禁止指示的 UART 中断源。只有使能的中断源才会反映到处理器中断，禁止的中断源对处理器无影响。其中，参数 ui32IntFlags 与 UARTIntEnable( ) 函数中的参数定义相同
返回参数	无

表 A-30　UARTIntEnable

功能	使能单个 UART 中断源
函数原型	void UARTIntEnable (uint32_t ui32Base, uint32_t ui32IntFlags)
参数	ui32Base　UART 端口的基地址 ui32IntFlags　使能中断源的位掩码
描述	该函数使能指示的 UART 中断源 参数 ui32IntFlags 为下列任意值的逻辑或: UART_INT_9BIT　//9 位地址匹配中断 UART_INT_OE　//溢出错误中断 UART_INT_BE　//中止错误中断 UART_INT_PE　//奇偶错误中断 UART_INT_FE　//帧错误中断 UART_INT_RT　//接收超时中断 UART_INT_TX　//发送中断 UART_INT_RX　//接收中断 UART_INT_DSR　//DSR 中断 UART_INT_DCD　//DCD 中断 UART_INT_CTS　//CTS 中断 UART_INT_RI　//RI 中断
返回参数	无

表 A-31　UARTIntRegister

功能	注册 UART 的中断处理程序
函数原型	void UARTIntRegister (uint32_t ui32Base, void (* pfnHandler) (void))
参数	ui32Base　UART 端口的基地址 pfnHandler　指向 UART 中断发生时被调函数的指针
描述	该函数注册中断处理程序。该函数使能中断控制器中的全局中断;特定的 UART 中断必须通过 UARTIntEnable ()函数使能。且由中断处理程序来负责清除中断源
返回参数	无

表 A-32　UARTIntStatus

功能	获取当前的中断状态
函数原型	uint32_t UARTIntStatus (uint32_t ui32Base, bool bMasked)
参数	ui32Base　UART 端口的基地址 bMasked　若所需的是原始中断状态,则 bMasked 为 false;若所需的是屏蔽的中断状态,则 bMasked 为 true
描述	该函数返回指定的 UART 中断状态。无论是原始中断状态还是允许中断反映到处理器的状态,都可以被返回
返回参数	返回当前中断状态,描述在 UARTIntEnable()中枚举位字段的值

表 A-33　UARTIntUnregister

功能	注销 UART 中断的中断处理程序
函数原型	void UARTIntUnregister (uint32_t ui32Base)

（续）

参数	ui32Base　UART 端口的基地址
描述	该函数注销中断处理程序，清除 UART 中断发生时被调用的处理程序。它也会关闭中断控制器中的中断，使中断处理程序不会被再调用
返回参数	无

表 A-34　UARTModemControlClear

功能	清除 DTR 和/或 RTS 调制解调器控制信号的状态
函数原型	void UARTModemControlClear（uint32_t ui32Base, uint32_t ui32Control）
参数	ui32Base　UART 端口的基地址 ui32Control　表示调制解调器控制位置位的位映射标志
描述	此函数清除 UART 中 DTR 或 RTS 调制解调器的握手输出状态 参数 ui32Control 为下列值的逻辑或： ➢ UART_OUTPUT_DTR　　//调制解调器的 DTR 控制信号 ➢ UART_OUTPUT_RTS　　//调制解调器的 RTS 控制信号
返回参数	无

表 A-35　UARTModemControlGet

功能	获取 DTR 和 RTS 调制解调器控制信号的状态
函数原型	void UARTModemControlClear（uint32_t ui32Base, uint32_t ui32Control）
参数	ui32Base　UART 端口的基地址
描述	该函数返回两个 UART 调制解调器控制信号（DTR、RTS）的当前状态
返回参数	返回握手输出信号的状态。它为 UART_OUTPUT_RTS 与 UART_OUTPUT_DTR 组合逻辑或的值

表 A-36　UARTModemControlSet

功能	设置 DTR 和/或 RTS 调制解调器控制信号的状态
函数原型	void UARTModemControlSet（uint32_t ui32Base, uint32_t ui32Control）
参数	ui32Base　UART 端口的基地址 ui32Control　表示调制解调器控制位被置位的位映射标志
描述	该函数配置 UART 中 DTR 或 RTS 调制解调器握手输出的状态 参数 ui32Control 为下列值的逻辑或： ➢ UART_OUTPUT_DTR　　//调制解调器的 DTR 控制信号 ➢ UART_OUTPUT_RTS　　//调制解调器的 RTS 控制信号
返回参数	无

表 A-37　UARTModemStatusGet

功能	获取 RI、DCD、DSR 和 CTS 调制解调器状态信号的状态
函数原型	uint32_t　UARTModemStatusGet（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址

（续）

描述	该函数返回 UART 中 4 个调制解调器状态信号（RI、DCD、DSR、CTS）的当前状态
返回参数	返回握手输出信号的状态。此值为 UART_INPUT_RI、UART_INPUT_DCD、UART_INPUT_CTS 和 UART_INPUT_DSR 的组合逻辑或

表 A-38　UARTParityModeGet

功能	获取正在使用中的校验类型
函数原型	uint32_t UARTParityModeGet（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数用于获取传输数据和希望的接收数据的校验的类型
返回参数	返回当前的奇偶校验设置。其取值为下列值之一： ➢ UART_CONFIG_PAR_NONE　//无校验位 ➢ UART_CONFIG_PAR_EVEN　//偶校验 ➢ UART_CONFIG_PAR_ODD　//奇校验 ➢ UART_CONFIG_PAR_ONE　//校验位始终为 1(可直接控制校验位) ➢ UART_CONFIG_PAR_ZERO　//校验位始终为 0(可直接控制校验位)

表 A-39　UARTParityModeSet

功能	设置校验类型
函数原型	void UARTParityModeSet（uint32_t ui32Base， uint32_t ui32Parity）
参数	ui32Base　UART 端口的基地址 ui32Parity　指定使用的校验类型
描述	该函数用于配置发送和希望接收的校验类型 参数 ui32Parity 的取值必须为下列值之一： ➢ UART_CONFIG_PAR_NONE　//无校验位 ➢ UART_CONFIG_PAR_EVEN　//偶校验 ➢ UART_CONFIG_PAR_ODD　//奇校验 ➢ UART_CONFIG_PAR_ONE　//校验位始终为 1（可直接控制校验位） ➢ UART_CONFIG_PAR_ZERO　//校验位始终为 0（可直接控制校验位）
返回参数	无

表 A-40　UARTRxErrorClear

功能	清除所有报告的接收器错误
函数原型	void UARTRxErrorClear（uint32_ t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数用于清除所有接收器错误标志的报告，这些错误标志可以通过 UARTRxErrorGet( ) 得到。如果使用溢出、帧错误、校验错误或打断中断，那么清除中断后必须调用此函数，以确保后面的同类型错误可触发另一次中断
返回参数	无

表 A-41　UARTRxErrorGet

功能	获取当前接收器错误
函数原型	uint32_t UARTRxErrorGet（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址

（续）

描述	该函数返回 4 个接收器错误源各自的当前状态。通过前一次调用 UARTCharGet( ) 或 UARTCharGetNonBlocking( )，返回的错误就相等于返回 4 个错误位。除非溢出错误被设置成立即发生，而不是在下一次读取一个字符时才发生
返回参数	返回接收错误标志为 UART_RXERROR_FRAMING、UART_RXERROR_PARITY、UART_RXERROR_BREAK、UART_RXERROR_OVERRUN）的组合逻辑或

表 A-42　UARTSmartCardDisable

功能	禁止指定 UART 的 ISO7816 智能卡模式
函数原型	void UARTSmartCardDisable（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数清除 UART 控制寄存器的 SMART 位（ISO7816 智能卡）
返回参数	无

表 A-43　UARTSmartCardEnable

功能	使能指定 UART 的 ISO7816 智能卡模式
函数原型	void UARTSmartCardEnable（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数使能 UART 中的 ISO7816 智能卡模式的 SMART 控制位 其中，ISO7816 要求字长为 8 位和偶校验
返回参数	无

表 A-44　UARTSpaceAvail

功能	确定发送 FIFO 中是否有可用的空间
函数原型	bool UARTSpaceAvail（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数返回一个指示发送 FIFO 是否有可用空间的标志
返回参数	如果发送 FIFO 中有可用的空间，则返回 true；如果发送 FIFO 中无可用的空间，则返回 false

表 A-45　UARTTxIntModeGet

功能	返回 UART 发送中断的当前操作模式
函数原型	uint32_t UARTTxIntModeGet（uint32_t ui32Base）
参数	ui32Base　UART 端口的基地址
描述	该函数返回 UART 发送中断的当前的操作模式 返回值如下： ➢ UART_TXINT_MODE_EOT //如果当前配置的发送中断有效，一旦发送器完全处于闲置状态，将返回 UART_TXINT_MODE_EOT ➢ UART_TXINT_MODE_FIFO //根据发送 FIFO 的深度，如果中断配置有效，将返回 UART_TXINT_MODE_FIFO
返回参数	返回 UART_TXINT_MODE_FIFO 或 UART_TXINT_MODE_EOT

表 A-46 UARTTxIntModeSet

功能	设置 UART 发送中断的操作模式
函数原型	void UARTTxIntModeSet (uint32_t ui32Base, uint32_t ui32Mode)
参数	ui32Base UART 端口的基地址 ui32Mode ui32Mode 是发送中断的操作模式。如果为 UART_TXINT_MODE_EOT，则当发送器为空时触发中断；如果为 UART_TXINT_MODE_FIFO，则根据发送 FIFO 的深度去触发中断
描述	函数允许设置 UART 传输中断的模式。默认情况下，当 FIFO 深度处于阈值内时可申请一个发送中断（调用 UARTFIFOLevelSet( )函数完成阈值设置）。或者，如果调用这个函数(在参数 ui32Mode 设置成 UART_TXINT_MODE_EOT 时)，则一旦发送器完全处于空闲状态，将发出一个发送中断请求
返回参数	无



附录 B 第 4 章附录：ADC 固件库函数简介

本章附录将简要介绍 ADC 固件库函数的参数定义及功能说明。

表 B-1 ADCBusy

功能	确定 ADC 是否为忙状态
函数原型	bool ADCBusy (uint32_t ui32Base)
参数	ui32Base ADC 模块基地址
描述	该函数允许调用者确定是否当前采样 ADC。如果返回 false，则 ADC 未在采样数据。让设备进入深度睡眠之前，使用此函数来检测 ADC 数据采样的完成。在使用该函数前，强烈建议将所有使能的序列发生器的事件触发改为 ADC_TRIGGER_NEVER，以防止在检查到忙状态后启动 ADC 采样
返回	如果 ADC 正在进行采样，则返回 true；如果 ADC 采样完成，才返回 false

表 B-2 ADCComparatorConfigure

功能	配置 ADC 的数字比较器
函数原型	void ADCComparatorConfigure (uint32_t ui32Base, uint32_t ui32Comp, uint32_t ui32Config)
参数	ui32Base ADC 模块基地址 ui32Comp 配置比较器的索引 ui32Config 比较器的配置
描述	该函数用于配置比较器。参数 ui32Config 为 ADC_COMP_TRIG_xxx 与 ADC_COMP_INT_xxx 的逻辑或 ① ADC_COMP_TRIG_xxx 可取以下的值： ➢ ADC_COMP_TRIG_NONE //不触发 PWM 故障条件 ➢ ADC_COMP_TRIG_LOW_ALWAYS //当 ADC 输出处于低频带时，将触发 PWM 故障条件 ➢ ADC_COMP_TRIG_LOW_ONCE //当 ADC 输出至低频带时，会触发一次 PWM 故障条件 ➢ ADC_COMP_TRIG_LOW_HALWAYS //仅当自上次的触发输出 ADC 输出已经在高频段，当 ADC 输出至低频段时，总是会触发 PWM 故障条件

描述	➢ ADC_COMP_TRIG_LOW_HONCE　//仅当自上次的触发输出 ADC 输出已经在高频段，当 ADC 输出至低频段时，会触发一次 PWM 故障条件 ➢ ADC_COMP_TRIG_MID_ALWAYS //当 ADC 输出处于中频段时，总是会触发 PWM 错误条件 ➢ ADC_COMP_TRIG_MID_ONCE　//当 ADC 输出转换至中频段时，将触发一次 PWM 错误条件 ➢ ADC_COMP_TRIG_HIGH_ALWAYS　//当 ADC 输出处于高频段时，总是触发 PWM 错误条件 ➢ ADC_COMP_TRIG_HIGH_ONCE　//当 ADC 输出至高频段时，会触发一次 PWM 错误条件 ➢ ADC_COMP_TRIG_HIGH_HALWAYS　//如果自上一次触发 ADC 输出已在低频段，当 ADC 输出至高频段时，总是会触发 PWM 错误条件 ➢ ADC_COMP_TRIG_HIGH_HONCE　//如果自上一次触发 ADC 输出已在低频段，当 ADC 输出至高频段时，将会触发一次 PWM 错误条件 ② ADC_COMP_INT_xxx 的取值如下： ➢ ADC_COMP_INT_NONE　//不产生 ADC 中断 ➢ ADC_COMP_INT_LOW_ALWAYS　//当 ADC 输出低频段时，将产生 ADC 中断 ➢ ADC_COMP_INT_LOW_ONCE　//当 ADC 输出低频段时，将会产生一次 ADC 中断 ➢ ADC_COMP_INT_LOW_HALWAYS　//如果自上一次触发 ADC 输出已在高频段，当 ADC 输出至低频段时，总是会产生 ADC 中断 ➢ ADC_COMP_INT_LOW_HONCE　//如果自上一次触发 ADC 输出已在高频段，当 ADC 输出至低频段时，将产生一次 ADC 中断 ➢ ADC_COMP_INT_MID_ALWAYS　//当 ADC 输出中频段时，将产生 ADC 中断 ➢ ADC_COMP_INT_MID_ONCE //当 ADC 输出中频段时，将产生一次 ADC 中断 ➢ ADC_COMP_INT_HIGH_ALWAYS　//当 ADC 输出高频段时，将产生 ADC 中断 ➢ ADC_COMP_INT_HIGH_ONCE　//当 ADC 输出高频段时，将会产生一次 ADC 中断 ➢ ADC_COMP_INT_HIGH_HALWAYS　//如果自上一次触发 ADC 输出已在低频段，当 ADC 输出至高频段时，总是会产生 ADC 中断 ➢ ADC_COMP_INT_HIGH_HONCE　//如果自上一次触发 ADC 输出已在低频段，只有当 ADC 输出至高频段时，将产生一次 ADC 中断
返回	无

表 B-3　ADCComparatorIntClear

功能	清除采样序列比较器的中断源
函数原型	void ADCComparatorIntClear（uint32_t ui32Base, uint32_t ui32Status）
参数	ui32Base　ADC 模块基地址 ui32Status　清除位映射的中断状态
描述	清除指定的中断状态
返回	无

表 B-4　ADCComparatorIntDisable

功能	禁止采样序列比较器中断
函数原型	void ADCComparatorIntDisable（uint32_t ui32Base, uint32_t ui32SequenceNum）
参数	ui32Base ADC　ADC 模块基地址 ui32SequenceNum　采样序列号
描述	该函数禁止请求采样序列比较器的中断
返回	无

表 B-5　ADCComparatorIntEnable

功能	使能采样序列比较器中断
函数原型	void ADCComparatorIntEnable（uint32_t　ui32Base, uint32_t ui32SequenceNum）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号
描述	该函数使能请求采样序列比较器的中断
返回	无

表 B-6　ADCComparatorIntStatus

功能	获得采样序列比较器中断状态
函数原型	uint32_t ADCComparatorIntStatus（uint32_t ui32Base）
参数	ui32Base　ADC 模块基地址
描述	该函数返回数字比较器中断状态位。该状态是序列未知的
返回	当前比较器的中断状态

表 B-7　ADCComparatorRegionSet

功能	定义 ADC 数字比较器区域
函数原型	void ADCComparatorRegionSet（uint32_t ui32Base, uint32_t ui32Comp, uint32_t ui32LowRef, uint32_t ui32HighRef）
参数	ui32Base　ADC 模块基地址 ui32Comp　比较器配置索引 ui32LowRef　低中频阈值基准点 ui32HighRef　高中频阈值基准点
描述	ADC 数字比较器的操作是基于三个 ADC 值的区域： low - band：表示 ADC 的值≤ui32LowRef 的值 mid - band：表示 ui32LowRef 值≤ADC 值≤ui32HighRef 值 high - band：表示 ADC 值≥ui32HighRef 的值
返回	无

表 B-8　ADCComparatorReset

功能	复位 ADC 数字比较器条件
函数原型	void ADCComparatorReset（uint32_t ui32Base, uint32_t ui32Comp, bool bTrigger, bool bInterrupt）
参数	ui32Base　ADC 模块基地址 ui32Comp　比较器索引 bTrigger　指示复位触发条件的标志 bInterrupt　指示复位中断条件的标志
描述	由于数字比较器要使用当前和以前的 ADC 值，该函数允许比较器复位到它的初始值以防止当一个序列使能时使用过时的数据
返回	无

表 B-9　ADCHardwareOversampleConfigure

功能	配置 ADC 硬件过采样因子
函数原型	void ADCHardwareOversampleConfigure（uint32_t ui32Base, uint32_t ui32Factor）
参数	ui32Base　ADC 模块的基地址 ui32Factor　待平均的采样数
描述	该函数为 ADC 配置硬件过采样因子，可用于提高采样数据的分辨率。过采样是通过平均多个样本相同的模拟输入。硬件过采样支持 2x、4x、8x、16x、32x 和 64x 六种过采样率，若指定过采样因子为 0，则将禁止硬件过采样。硬件过采样的配置对所有采样序列都有效。硬件过采样不会像软件过采样那样，降低采样序列的深度；每个写入采样序列发生器 FIFO 中的采样值是完全过采样的模拟输入读数 使能硬件平均采样值虽然可提高 ADC 的精确度，但是以增加采样的吞吐量为代价的
返回	无

表 B-10　ADCIntClear

功能	清除采样序列的中断源
函数原型	void ADCIntClear（uint32_t ui32Base, uint32_t ui32SequenceNum）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号
描述	清除指定采样序列的中断源，使其失效。该函数必须在中断处理程序中调用，以防止在退出时再次立即触发中断
返回	无

表 B-11　ADCIntClearEx

功能	清除指定 ADC 中断源
函数原型	void　ADCIntClearEx（uint32_t ui32Base, uint32_t ui32IntFlags）
参数	ui32Base　ADC 模块基地址 ui32IntFlags　禁止中断源的位掩码
描述	清除指定中断源的中断。其中，ui32IntFlags 参数是 ADC_INT_ * 值的逻辑或
返回	无

表 B-12　ADCIntDisable

功能	禁止采样序列中断
函数原型	void ADCIntDisable（uint32_t ui32Base, uint32_t ui32SequenceNum）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号
描述	该函数用来禁止请求的采样序列中断
返回	无

表 B-13　ADCIntDisableEx

功能	禁止 ADC 中断源
函数原型	void　ADCIntDisableEx（uint32_t ui32Base, uint32_t ui32IntFlags）

（续）

参数	ui32Base　ADC 模块基地址 ui32IntFlags　禁止中断源的位掩码
描述	该函数禁止指定的 ADC 中断源 参数 ui32IntFlags 为任意下值的逻辑或： ➢ ADC_INT_SS0 //ADC 采样序列 0 中断 ➢ ADC_INT_SS1 ➢ ADC_INT_SS2 ➢ ADC_INT_SS3 ➢ ADC_INT_DMA_SS0 //DMA 对 ADC 采样序列 0 的中断 ➢ ADC_INT_DMA_SS1 ➢ ADC_INT_DMA_SS2 ➢ ADC_INT_DMA_SS3 ➢ ADC_INT_DCON_SS0 //数字比较器对 ADC 采样序列 0 的中断 ➢ ADC_INT_DCON_SS1 ➢ ADC_INT_DCON_SS2 ➢ ADC_INT_DCON_SS3
返回	无

表 B-14　ADCIntEnable

功能	使能 ADC 采样序列中断
函数原型	void ADCIntEnable（uint32_t ui32Base, uint32_t　ui32SequenceNum）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号
描述	该函数使能请求采样序列的中断。任何未完成的中断在使能采样序列中断前将被清除
返回	无

表 B-15　ADCIntEnableEx

功能	使能 ADC 的中断源
函数原型	void　ADCIntEnableEx（uint32_t ui32Base, uint32_t ui32IntFlags）
参数	ui32Base　ADC 模块基地址 ui32IntFlags　禁止中断源的位掩码
描述	该函数使能指定的 ADC 中断源 ui32IntFlags 参数以下任意组合的逻辑或： ➢ ADC_INT_SS0　　//ADC 采样序列 0 中断 ➢ ADC_INT_SS1 ➢ ADC_INT_SS2 ➢ ADC_INT_SS3 ➢ ADC_INT_DMA_SS0　　//DMA 对 ADC 采样序列 0 的中断 ➢ ADC_INT_DMA_SS1 ➢ ADC_INT_DMA_SS2 ➢ ADC_INT_DMA_SS3 ➢ ADC_INT_DCON_SS0　　//数字比较器对 ADC 采样序列 0 的中断 ➢ ADC_INT_DCON_SS1 ➢ ADC_INT_DCON_SS2 ➢ ADC_INT_DCON_SS3
返回	无

表 B-16　ADCIntRegister

功能	注册 ADC 中断的中断处理程序
函数原型	void ADCIntRegister（uint32_t ui32Base, uint32_t ui32SequenceNum, void（*pfnHandler）（void））
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号 pfnHandler　指向 ADC 中断处理函数的函数指针
描述	该函数设置采样序列中断发生时被调用的处理程序。该函数使能中断控制器的全局变中断；序列中断必须用 ADCIntEnable( )函数使能。中断处理程序使用 ADCIntClear( )函数负责清除中断源
返回	无

表 B-17　ADCIntStatus

功能	获得当前中断状态
函数原型	uint32_t　ADCIntStatus（uint32_t ui32Base, uint32_t ui32SequenceNum, bool bMasked）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号 bMasked　如果需要原始中断状态，则 bMasked 为 false；如果需要屏蔽中断状态，则 bMasked 为 true
描述	这个函数返回指定的采样序列中断状态。无论是原始中断状态，还是允许反映到处理器的中断状态均可返回
返回	当前原始/屏蔽中断状态

表 B-18　ADCIntStatusEx

功能	获得指定 ADC 模块的当前中断状态
函数原型	uint32_t　ADCIntStatusEx（uint32_t ui32Base，bool bMasked）
参数	ui32Base　ADC 模块基地址 bMasked　指定返回的是屏蔽中断状态还是原始中断状态
描述	若 bMasked 设置为 true，则返回屏蔽中断状态；否则返回原始中断状态
返回	指定 ADC 模块的当前中断状态。返回值是当前活动 ADC_INT_* 值的逻辑或

表 B-19　ADCIntUnregister

功能	注销 ADC 中断的中断处理程序
函数原型	void ADCIntUnregister（uint32_t ui32Base, uint32_t ui32SequenceNum）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号
描述	注销 ADC 中断处理程序。该函数禁止中断控制器中的全局中断；必须通过 ADCIntDisable( )来禁止序列中断
返回	无

表 B-20　ADCPhaseDelayGet

功能	获取触发与序列开始之间的相位延迟
函数原型	uint32_t ADCPhaseDelayGet（uint32_t ui32Base）
参数	ui32Base　ADC 模块基地址
描述	该函数获取当前 ADC 检测到触发事件与序列开始之间的相位延迟
返回	返回的相位延迟为下列其中一个： ➢ ADC_PHASE_0 ➢ ADC_PHASE_22_5 ➢ ADC_PHASE_45 ➢ ADC_PHASE_67_5 ➢ ADC_PHASE_90 ➢ ADC_PHASE_112_5 ➢ ADC_PHASE_135 ➢ ADC_PHASE_157_5 ➢ ADC_PHASE_180 ➢ ADC_PHASE_202_5 ➢ ADC_PHASE_225 ➢ ADC_PHASE_247_5 ➢ ADC_PHASE_270 ➢ ADC_PHASE_292_5 ➢ ADC_PHASE_315 ➢ ADC_PHASE_337_5

表 B-21　ADCPhaseDelaySet

功能	设置的触发与序列开始之间的相位延迟
函数原型	void ADCPhaseDelaySet（unsigned long ulBase, unsigned long ulPhase）
参数	ui32Base　ADC 模块基地址 ui32Phase　指定的相位延迟取值为下列中的一个： ➢ ADC_PHASE_0 ➢ ADC_PHASE_22_5 ➢ ADC_PHASE_45 ➢ ADC_PHASE_67_5 ➢ ADC_PHASE_90 ➢ ADC_PHASE_112_5 ➢ ADC_PHASE_135 ➢ ADC_PHASE_157_5 ➢ ADC_PHASE_180 ➢ ADC_PHASE_202_5 ➢ ADC_PHASE_225 ➢ ADC_PHASE_247_5 ➢ ADC_PHASE_270 ➢ ADC_PHASE_292_5 ➢ ADC_PHASE_315 ➢ ADC_PHASE_337_5
描述	该函数设置 ADC 检测到触发事件与序列开始之间的相位延迟。通过选择不同的相位延迟（如 ADC_PHASE_0 与 ADC_PHASE_180），使每个 ADC 模块可以对同一个模拟输入端进行采样，可增加模拟输入的采样率。该 ADC 模块对于该模块内的所有采样序列都具有单一的相位延迟
返回	无

表 B-22　ADCProcessorTrigger

功能	导致采样序列的处理器触发
函数原型	void ADCProcessorTrigger（uint32_t ui32Base, uint32_t ui32SequenceNum）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号，为 ADC_TRIGGER_WAIT 或 ADC_TRIGGER_SIGNAL 的任意或运算
描述	在采样序列的触发配置成 ADC_TRIGGER_PROCESSOR 时，该函数可用来触发一个启动处理器的采样序列。如果序列号由 ADC_TRIGGER_WAIT 的或运算得到，则启动处理器的触发将被延迟，直到稍后启动处理器的触发指定为 ADC_TRIGGER_AIGNAL 的一个不同 ADC 单元，允许多个 ADC 以同步方式从启动处理器的触发开始采样
返回	无

表 B-23　ADCReferenceGet

功能	返回当前设置的 ADC 基准
函数原型	uint32_t ADCReferenceGet（uint32_t ui32Base）
参数	ui32Base　ADC 模块基地址
描述	返回设置的 ADC 基准电压值。其返回值如下： ➢ ADC_REF_INT ➢ ADC_REF_EXT_3V ➢ ADC_REF_EXT_1V
返回	当前设置的 ADC 基准电压

表 B-24　ADCReferenceSet

功能	选择 ADC 基准电压
函数原型	void ADCReferenceSet（uint32_t ui32Base, uint32_t ui32Ref）
参数	ui32Base ADC　模块基地址 ui32Ref　使用的基准
描述	用 ui32Ref 参数指定 ADC 的基准电压。 其取值必须为以下列出的内部或外部基准电压之一： ➢ ADC_REF_INT ◇ ADC_REF_EXT_3V ◇ ADC_REF_EXT_1V ➢ 如果选择 ADC_REF_INT，则使用内部的基准 3V 电压，无须外部基准电压 ➢ 若选择 ADC_REF_EXT_3V，则 3V 的基准电压必须连接到 VDD 引脚 ➢ 若选择 ADC_REF_EXT_1V，则 1V 的外部基准电压必须连接到 VDD 引脚
返回	无

表 B-25　ADCSequenceConfigure

功能	配置采样序列触发源和优先级
函数原型	void ADCSequenceConfigure（ uint32_t　ui32Base, uint32_t　ui32SequenceNum, uint32_t　ui32Trigger, uint32_t　ui32Priority）

（续）

参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号 ui32Trigger　启动采样序列的触发源；必须是 ADC_TRIGGER_ * 的值之一 ui32Priority　相对于其他采样序列的相对优先级
描述	此函数配置采样序列的初始标准。有效的采样序列范围为 0 ~ 3；序列发生器 0 将捕获多达 8 个采样值、序列发生器 1 和 2 可捕获四个采样值、序列发生器 3 仅捕获单个采样。触发条件和优先级（相对于其他采样序列执行）被设置 ulTrigger 参数的取值如下： ➢ ADC_TRIGGER_PROCESSOR //由处理器通过 ADCProcessorTrigger（）函数产生触发 ➢ ADC_TRIGGER_COMP0　//配置 ComparatorConfigure（）函数，由第一个模拟比较器产生触发 ➢ ADC_TRIGGER_COMP1 //配置 ComparatorConfigure（）函数，由第二个模拟比较器产生触发 ➢ ADC_TRIGGER_COMP2 //配置 ComparatorConfigure（）函数，由第三个模拟比较器产生触发 ➢ ADC_TRIGGER_EXTERNAL //由端口 B4 的引脚输入产生触发。注意：一些微控制器可以使用 GPIOADCTriggerEnable（）函数选择任意 GPIO ➢ ADC_TRIGGER_TIMER　//配置与 TimerCon trolTrigger（）函数，由定时器产生触发 ➢ ADC_TRIGGER_PWM0 //配置 PWMGenIntTrigEnable（），由第一个 PWM 发生器产生触发 ➢ ADC_TRIGGER_PWM1 //配置 PWMGenIntTrigEnable（），由第二个 PWM 发生器产生触发 ➢ ADC_TRIGGER_PWM2 //配置 PWMGenIntTrigEnable（），由第三个 PWM 发生器产生触发 ➢ ADC_TRIGGER_PWM3　//配置 PWMGenIntTrigEnable（），由第四个 PWM 发生器产生触发 ➢ ADC_TRIGGER_ALWAYS －触发总是有效，造成重复捕捉采样序列（只要没有更高的优先级源有效）
返回	无

表 B-26　ADCSequenceDataGet

功能	获取采样序列的捕获数据
函数原型	int32_t　ADCSequenceDataGet（uint32_t ui32Base, uint32_t　ui32SequenceNum, uint32_t　* pui32Buffer）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号 pui32Buffer　存放数据的地址
描述	该函数把指定采样序列发生器输出 FIFO 中的数据复制到内存驻留缓冲区中。硬件 FIFO 中的采样数据将被复制到缓冲区中（假定缓冲区足够容纳这些数据）。如果正在执行这些采样，这可能不是整个采样序列，因此函数只返回当前可用的采样数据
返回	返回复制到缓冲区中的采样数

表 B-27　ADCSequenceDisable

功能	禁止采样序列
函数原型	void ADCSequenceDisable（uint32_t ui32Base, uint32_t ui32SequenceNum）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号
描述	当检测到触发时，防止指定的采样序列被捕获。配置采样序列前应禁止该采样序列
返回	无

表 B-28　ADCSequenceDMADisable

功能	禁止采样序列发生器的 DMA
函数原型	void　ADCSequenceDMADisable（uint32_t ui32Base, uint32_t ui32SequenceNum）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号
描述	禁止产生 DMA 请求的指定序列发生器采样
返回	无

表 B-29　ADCSequenceDMAEnable

功能	使能采样序列发生器的 DMA
函数原型	void　ADCSequenceDMAEnable（uint32_t ui32Base, uint32_t ui32SequenceNum）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号
描述	根据采样序列发生器的 FIFO 深度允许产生 DMA 请求
返回	无

表 B-30　ADCSequenceEnable

功能	使能采样序列
函数原型	void ADCSequenceEnable（uint32_t ui32Base, uint32_t ui32SequenceNum）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号
描述	当检测到触发时，允许指定的采样序列被捕获。在采样序列使能前必须首先对其进行配置
返回	无

表 B-31　ADCSequenceOverflow

功能	确定采样序列是否发生溢出
函数原型	int32_t　ADCSequenceOverflow（uint32_t　ui32Base, uint32_t　ui32SequenceNum）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号
描述	该函数确定采样序列是否发生溢出。如果捕获到的采样数据在下一次触发前没有从 FIFO 中读取，将会产生溢出
返回	无溢出，返回 0；有溢出，返回非 0

表 B-32　ADCSequenceOverflowClear

功能	清除采样序列的溢出条件
函数原型	void ADCSequenceOverflowClear（uint32_t ui32Base, uint32_t ui32SequenceNum）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号
描述	该函数清除采样序列的一个溢出条件。为了检测后续溢出条件，必须清除溢出条件
返回	无

表 B-33　ADCSequenceStepConfigure

功能	配置采样序列发生器的步
函数原型	void ADCSequenceStepConfigure（uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t ui32Step, uint32_t ui32Config）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号 ui32Step　配置步 ui32Config　该步的配置是下列取值的逻辑或： ➢ ADC_CTL_TS　　//内部温度传感器 ➢ ADC_CTL_IE　　//中断使能 ➢ ADC_CTL_END　//采样结束 ➢ ADC_CTL_CHx　//采样输入通道 ➢ ADC_CTL_D　　//差分选择 ➢ ADC_CTL_TS　　//内部温度传感器 ➢ ADC_CTLCMPx　//发送 ADC 采样值到选定的比较器
描述	该函数用于配置 ADC 采样序列的步。ADC 可以配置成单端或差分操作（将 ADC_CTL_D 位置位来选择差分操作）；选择被采样的通道（ADC_CTL_CH0 ~ ADC_CTL_CH23 的值）；以及选择内部温度传感器（ADC_CTL_TS 位） 此外，该步可定义为序列的末端（ADC_CTL_END 位），可将其配置成在步完成之后，发出一个中断（ADC_CTL_IE 位） 当设备上包含数字比较器时，也可使用 ADC_CTL_CMP0 ~ ADC_CTL_CMP7 来将此步配置成发送 ADC 采样到所选比较器。在该序列的触发发生时，ADC 将会在恰当的时间使用该配置
返回	无

表 B-34　ADCSequenceUnderflow

功能	确定采样序列是否发生下溢出
函数原型	int32_t　ADCSequenceUnderflow（uint32_t　ui32Base, uint32_t ui32SequenceNum）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号
描述	该函数确定一个采样序列是否发生下溢出。如果从 FIFO 中读出的采样值太多时，将会发生下溢
返回	无下溢，则返回 0；发生下溢出，将返回非 0

表 B-35　ADCSequenceUnderflowClear

功能	清除采样序列的下溢出条件
函数原型	void　ADCSequenceUnderflowClear（uint32_t　ui32Base, uint32_t　ui32SequenceNum）
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号
描述	该函数将清除采样序列中的一个下溢条件。为了检测随后的下溢条件，必须清除该下溢条件
返回	无

表 B-36　ADCSoftwareOversampleConfigure

功能	配置 ADC 软件的过采样因子
函数原型	void ADCSoftwareOversampleConfigure (uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t ui32Factor)
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号 ui32Factor　采样数据的平均值
描述	该函数配置 ADC 的软件过采样，它可以提高采样数据的分辨率。软件过采样对同一模拟通道输入的多个采样值取平均以提高采样值的精度。软件过采样支持三种（2x、4x、8x）的过采样速率 过采样仅支持深度超过一个采样的采样序列发生器（即第 4 个采样序列发生器不被支持，因为其只有一个采样深度）
返回	无

表 B-37A　DCSoftwareOversampleDataGet

功能	使用软件过采样获取采样序列的捕获数据
函数原型	void ADCSoftwareOversampleDataGet (uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t *pui32Buffer, uint32_t ui32Count)
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号 pui32Buffer　存放数据的地址 ui32Count　读取的采样数
描述	该函数应用软件过采样将指定采样序列输出 FIFO 中的数据复制到驻留内存的缓冲区中。所需的采样值被复制到数据缓冲区中；如果在硬件 FIFO 中没有足够的采样值以满足这些过采样数据项，将返回不正确的结果。调用者仅读取可获取的采样值，将一直等待到有足够的可用数据
返回	无

表 B-38　ADCSoftwareOversampleStepConfigure

功能	配置软件过采样序列发生器的步
函数原型	void　ADCSoftwareOversampleStepConfigure (uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t ui32Step, uint32_t ui32Config)
参数	ui32Base　ADC 模块基地址 ui32SequenceNum　采样序列号 ui32Step　配置步值 ui32Config　此步的配置
描述	使用软件过采样功能时，该函数用于配置采样序列发生器的步。可用的步数取决于 ADCSoftwareOversampleConfigure()函数配置的过采样因子。其中，参数 ui32Config 的值与 ADCSequenceStepConfigure()中所定义的值相同
返回	无



附录 C 第 5 章附录：GPIO 固件库函数简介

本章附录将简要介绍 GPIO 固件函数的参数定义及功能说明。

表 C-1 GPIOADCTriggerDisable

功能	禁止 GPIO 引脚作为启动 ADC 捕获的触发
函数原型	void GPIOADCTriggerDisable（uint32_t ui32Port, uint8_t ui8Pins）
参数	ui32Port GPIO 端口的基地址 ui8Pins 引脚的位填充（bit - packed）
描述	该函数禁止 GPIO 引脚作为启动 ADC 序列的触发。该函数可用于禁止通过调用 GPIOADCTriggerEnable（）使能的动作
返回参数	无

表 C-2 GPIOADCTriggerEnable

功能	使能 GPIO 引脚作为启动 ADC 捕获的触发
函数原型	void GPIOADCTriggerEnable（uint32_t ui32Port, uint8_t ui8Pins）
参数	ui32Port GPIO 端口的基地址 ui8Pins 引脚的位填充（bit - packed）
描述	该函数使能 GPIO 引脚作为启动 ADC 序列的触发。任何 GPIO 引脚可被配置成 ADC 序列的一个外部触发。如果使能所选引脚上的中断，GPIO 引脚将产生中断。若要使用一个 GPIO 引脚来触发 ADC 模块，必须用参数 ADC_TRIGGER_EXTERNAL 调用 ADCSequenceConfigure( ) 函数
返回参数	无

表 C-3 GPIODirModeGet

功能	获取引脚的方向和模式
函数原型	uint32_t GPIODirModeGet（uint32_t ui32Port, uint8_t ui8Pin）
参数	ui32Port GPIO 端口的基地址 ui8Pin 引脚号
描述	该函数获取所选 GPIO 端口上指定引脚的方向和控制模式。在软件控制下该引脚既可配置为输入也可配置为输出，或者引脚由硬件控制。控制类型和方向作为枚举数据类型返回
返回参数	为 GPIODirModeSet( ) 返回一个描述的枚举数据类型

表 C-4 GPIODirModeSet

功能	设置指定引脚的方向和模式
函数原型	voidGPIODirModeSet（uint32_t ui32Port, uint8_t ui8Pins, uint32_t ui32PinIO）
参数	ui32Port GPIO 端口的基地址 ui8Pins 引脚的位填充（bit - packed） ui32PinIO 引脚的方向和/或模式

（续）

描述	在软件控制下，该函数既可把所选 GPIO 端口上指定的引脚配置成输入也可配置为输出，或者将引脚配置成由硬件控制 参数 ui32PinIO 是一个枚举数据类型，其值为下列之一： ➢ GPIO_DIR_MODE_IN　　//指定引脚作为软件编程控制的输入 ➢ GPIO_DIR_MODE_OUT　//指定引脚作为软件编程控制的输出 ➢ GPIO_DIR_MODE_HW　　//指定引脚由硬件控制 使用位填充字节来指定引脚，其中被置位的每个位，标识要访问的引脚，字节的位 0 表示 GPIO 端口引脚 0，位 1 表示 GPIO 端口引脚 1，依此类推
返回参数	无

表 C-5　GPIODMATriggerDisable

功能	禁止 GPIO 引脚作为启动 DMA 交换的触发引脚
函数原型	void GPIODMATriggerDisable（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit－packed）
描述	该函数禁止 GPIO 引脚作为启动 DMA 交换的触发引脚。此函数可禁止通过调用 GPIODMATriggerEnable( )函数使能的 uDMA 交换
返回参数	无

表 C-6　GPIODMATriggerEnable

功能	使能 GPIO 引脚作为启动 DMA 交换的触发引脚
函数原型	void GPIODMATriggerEnable（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit－packed）
描述	该函数使能 GPIO 引脚作为启动 uDMA 交换的触发引脚。任何 GPIO 引脚都可配置成外部触发 uDMA。如果已使能了所选引脚的中断，则 GPIO 引脚将产生中断
返回参数	无

表 C-7　GPIOIntClear

功能	清除指定的中断源
函数原型	void GPIOIntClear(uint32_t　ui32Port, uint32_t　ui32IntFlags)
参数	ui32Port　GPIO 端口的基地址 ui32IntFlags　禁止中断源的位屏蔽
描述	清除指定中断源的中断 参数 ui32IntFlags 为 GPIO_INT_* 的逻辑或
返回参数	无

表 C-8　GPIOIntDisable

功能	禁止指定的 GPIO 中断
函数原型	void GPIOIntDisable(uint32_t　ui32Port, uint32_t　ui32IntFlags)
参数	ui32Port　GPIO 端口的基地址 ui32IntFlags　禁止中断源的位屏蔽

描述	该函数禁止指定的 GPIO 中断源。只有使能的中断源方能反映到处理器中断，禁止的中断源对处理器不会产生影响 ui32IntFlags 参数为以下值的逻辑或： ➢ GPIO_INT_PIN_0　　//引脚 0 中断 ➢ GPIO_INT_PIN_1 ➢ GPIO_INT_PIN_2 ➢ GPIO_INT_PIN_3 ➢ GPIO_INT_PIN_4 ➢ GPIO_INT_PIN_5 ➢ GPIO_INT_PIN_6 ➢ GPIO_INT_PIN_7　　//引脚 7 中断
返回参数	无

表 C-9　GPIOIntEnable

功能	使能指定的 GPIO 中断
函数原型	voidGPIOIntEnable（uint32_t　ui32Port， uint32_t　ui32IntFlags）
参数	ui32Port　GPIO 端口的基地址 ui32IntFlags　使能中断源的位屏蔽
描述	该函数使能指定中断源的中断 ui32IntFlags 参数是以下的逻辑或： ➢ GPIO_INT_PIN_0//引脚 0 中断 ➢ GPIO_INT_PIN_1 ➢ GPIO_INT_PIN_2 ➢ GPIO_INT_PIN_3 ➢ GPIO_INT_PIN_4 ➢ GPIO_INT_PIN_5 ➢ GPIO_INT_PIN_6 ➢ GPIO_INT_PIN_7　//引脚 7 中断
返回参数	无

表 C-10　GPIOIntRegister

功能	注册一个 GPIO 端口的中断处理程序
函数原型	void GPIOIntRegister（uint32_t　ui32Port， void（ * pfnIntHandler）（void））
参数	ui32Port　GPIO 端口的基地址 pfnIntHandler　指向 GPIO 端口中断处理函数的指针
描述	当从所选择的 GPIO 端口检测到一个中断时，此函数可确保调用由参数 pfnIntHandler 指定的中断处理程序。这个函数也可在中断控制器中使能相应 GPIO 的中断；单个引脚中断和中断源必须由 GPIOIntEnable( ) 函数使能
返回参数	无

表 C-11　GPIOIntStatus

功能	获取指定的 GPIO 端口中断状态
函数原型	uint32_t　GPIOIntStatus（uint32_t　ui32Port， bool　bMasked）
参数	ui32Port　GPIO 端口的基地址 bMasked　指定是否返回屏蔽或原始中断状态

（续）

描述	如果 bMasked 设置为 true，则返回屏蔽中断状态；否则，返回原始中断状态
返回参数	返回指定 GPIO 的模块的当前中断状态。返回当前被激活 GPIO_INT_* 值的逻辑或

表 C-12　GPIOIntTypeGet

功能	获取引脚的中断类型
函数原型	uint32_t　GPIOIntTypeGet（uint32_t　ui32Port, uint8_t　ui8Pin）
参数	ui32Port　GPIO 端口的基地址 ui8Pin　引脚号
描述	该函数获取被选 GPIO 端口上指定引脚上的中断类型。引脚可被配置成下降沿和上升沿、双边缘检测中断、高/低电平检测中断。中断检测机制的类型可被返回，并且可包括 GPIO_DISCRETE_INT 标志
返回参数	返回一个描述 GPIOIntTypeSet( ) 的标志

表 C-13　GPIOIntTypeSet

功能	设置指定引脚的中断类型
函数原型	void　GPIOIntTypeSet（uint32_t　ui32Port, uint8_ t　ui8Pins, uint32_t　ui32IntType）
参数	ui32Port　GPIO 端口的基地址 ui8Pin　引脚的位填充（bit - packed） ui32IntType　指定类型的中断触发机制
描述	该函数为被选 GPIO 端口上指定引脚设置各种中断的触发机制 ui32IntType 参数可用下列之一的标志定义： ➢ GPIO_FALLING_EDGE　//设置下降缘触发 ➢ GPIO_RISING_EDGE　//设置上升沿缘触发 ➢ GPIO_BOTH_EDGES　//设置双边缘触发 ➢ GPIO_LOW_LEVEL　//设置低电平触发 ➢ GPIO_HIGH_LEVEL　//设置高电平触发 除了上述标志外，也可用下列标志的或运算表示 ui32IntType 参数： ➢ GPIO_DISCRETE_INT　//设置 GPIO 端口每个引脚上的离散中断 这个 GPIO_DISCRETE_INT 并非适用于所有器件或所有 GPIO 端口，至于哪些设备和 GPIO 端口支持离散中断，请查看数据手册
返回参数	无

表 C-14　GPIOIntUnregister

功能	删除 GPIO 端口的中断处理程序
函数原型	void GPIOIntUnregister（uint32_t　ui32Port）
参数	ui32Port　GPIO 端口的基地址
描述	该函数注销指定 GPIO 端口的中断服务程序。该函数也禁止在中断控制器中相应 GPIO 端口的中断；对于单个的 GPIO 中断和中断源必须通过 GPIOIntDisable( ) 函数来禁止
返回参数	无

表 C-15　GPIOPadConfigGet

功能	获取引脚的焊盘（pad）配置
函数原型	void GPIOPadConfigGet（uint32_t　ui32Port, uint8_t　ui8Pin, uint32_t　* pui32Strength, uint32_t　* pui32PinType）
参数	ui32Port　GPIO 端口的基地址 ui8Pin　引脚号 pui32Strength　指向输出驱动强度存放单元的指针 pui32PinType　指向输出驱动类型存放单元的指针
描述	该函数获取被选 GPIO 端口上指定引脚的 pad ( ) 配置。在 pui32Strength 和 pui32PinType 的返回值应与在 GPIOPadConfigSet( ) 中使用的值对应。函数也可将引脚配置成输入引脚；然而，返回唯一有意义的数据为引脚终端是连接到一个上拉电阻还是下拉电阻上
返回参数	无

表 C-16　GPIOPadConfigSet

功能	设置指定的引脚的焊盘（pad）配置
函数原型	void GPIOPadConfigSet（uint32_t　ui32Port, uint8_t　ui8Pins, uint32_t　ui32Strength, uint32_t　ui32PinType）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit - packed） ui32Strength　指定输出驱动强度 ui32PinType　指定引脚类型
描述	该函数设置选定 GPIO 端口上指定引脚的驱动强度和类型。对于配置为输入端口的引脚，Pad 按要求配置，但唯一对输入有真正影响是上拉或下拉的终端配置 参数 ui32Strength 为以下值之一： ➢ GPIO_STRENGTH_2MA　//指定 2 mA 输出驱动 ➢ GPIO_STRENGTH_4MA ➢ GPIO_STRENGTH_8MA ➢ GPIO_STRENGTH_8MA_SC　//指定 8 mA 输出驱动带斜率控制 ➢ GPIO_STRENGTH_6MA ➢ GPIO_STRENGTH_10MA ➢ GPIO_STRENGTH_12MA 参数 ui32PinType 为以下值之一： ➢ GPIO_PIN_TYPE_STD　//推挽引脚 ➢ GPIO_PIN_TYPE_STD_WPU　//弱上拉引脚 ➢ GPIO_PIN_TYPE_STD_WPD　//弱下拉引脚 ➢ GPIO_PIN_TYPE_OD　//开漏引脚 ➢ GPIO_PIN_TYPE_ANALOG　//模拟输入 ➢ GPIO_PIN_TYPE_WAKE_HIGH　//高电平唤醒 ➢ GPIO_PIN_TYPE_WAKE_LOW　//低电平唤醒
返回参数	无

表 C-17　GPIOPinConfigure

功能	配置 GPIO 引脚的复用功能
函数原型	void GPIOPinConfigure（uint32_t　ui32PinConfig）
参数	ui32PinConfig　引脚的配置值，指定为唯一的 GPIO_P?? _??? 值

（续）

描述	该函数配置选定 GPIO 引脚复用的外设功能，每一个 GPIO 引脚一次只能配置成一种外设功能
返回参数	无

表 C-18　GPIOPinRead

功能	读取指定引脚的值
函数原型	int32_t　GPIOPinRead（uint32_t　ui32Port, uint8_t ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit - packed）
描述	读取被 ui8Pins 参数所指定引脚的值。返回输入和输出引脚的值，没有被 ui8Pins 指定的引脚被设置为 0
返回参数	返回一个位填充字节，其提供了指定引脚的状态。没有被 ui8Pins 指定的任何位返回 0。并且忽略 31:8 位

表 C-19　GPIOPinTypeADC

功能	配置作为模拟/数字转换器的输入引脚
函数原型	void GPIOPinTypeADC（uint32_t　ui32Port, uint8_t ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit - packed）
描述	模拟/数字转换器的输入引脚必须正确配置才能使其功能正常。该函数为这些引脚提供了正确的配置
返回参数	无

表 C-20　GPIOPinTypeCAN

功能	配置作为 CAN 设备的引脚
函数原型	void GPIOPinTypeCAN（uint32_t　ui32Port, uint8_t ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的 bit - packed（位填充）
描述	只有正确配置 CAN 引脚才能使 CAN 外设功能正常。此函数为这些引脚提供了一个典型的配置；其他配置可能取决于板上的设置
返回参数	无

表 C-21　GPIOPinTypeCIR

功能	配置作为用户红外线输入或输出的引脚
函数原型	void　GPIOPinTypeCIR（uint32_t ui32Port, uint8_t ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的 bit - packed（位填充）
描述	只有正确配置用户红外线引脚才能使该外设功能正常。此函数为这些引脚提供了一个正确的配置
返回参数	无

表 C-22　GPIOPinTypeComparator

功能	配置作为模拟比较器输入的引脚
函数原型	void　GPIOPinTypeComparator（uint32_t ui32Port, uint8_t ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit - packed）
描述	只有正确配置模拟比较器输入的引脚才能使该功能正常。此函数为这些引脚提供了正确的配置
返回参数	无

表 C-23　GPIOPinTypeEPI

功能	配置用于外设接口的引脚
函数原型	void GPIOPinTypeEPI（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit - packed）
描述	只有正确配置外设接口的引脚才能使该功能正常。此函数为这些引脚提供了一个典型的配置；其他配置可能取决于板上的设置
返回参数	无

表 C-24　GPIOPinTypeEthernetLED

功能	配置用于作为以太网外设 LED 信号的引脚
函数原型	void GPIOPinTypeEthernetLED（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit - packed）
描述	以太网外设提供两路用于驱动一个 LED 的信号（例如，链路状态/活动）。此函数为这些引脚提供了一个典型的配置
返回参数	无

表 C-25　GPIOPinTypeEthernetMII

功能	配置用于作为以太网外设 MII 信号的引脚
函数原型	void GPIOPinTypeEthernetMII（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit - packed）
描述	以太网外设上提供了一组 MII 信号用于连接到一个外部 PHY。此函数为这些引脚提供了一个典型的配置
返回参数	无

表 C-26　GPIOPinTypeFan

功能	配置用于通过风扇（fan）模块的引脚
函数原型	void GPIOPinTypeFan（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充 d（bit - packe）

（续）

描述	只有正确配置风扇引脚才能使该功能正常。此函数为这些引脚提供了一个典型的配置；其他配置可能取决于板上的设置
返回参数	无

表 C-27　GPIOPinTypeGPIOInput

功能	配置作为 GPIO 输入的引脚
函数原型	void GPIOPinTypeGPIOInput（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit－packed）
描述	只有正确配置作为 GPIO 引脚才能使该功能正常。此函数为这些引脚提供了正确的配置
返回参数	无

表 C-28　GPIOPinTypeGPIOOutput

功能	配置作为 GPIO 输出的引脚
函数原型	void GPIOPinTypeGPIOOutput（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit－packed）
描述	只有正确配置 GPIO 引脚才能使该功能正常。此函数为这些引脚提供了正确的配置
返回参数	无

表 C-29　GPIOPinTypeGPIOOutputOD

功能	配置作为 GPIO 开漏输出的引脚
函数原型	void GPIOPinTypeGPIOOutputOD（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit－packed）
描述	只有正确配置 GPIO 引脚才能使该功能正常。此函数为这些引脚提供了正确的配置
返回参数	无

表 C-30　GPIOPinTypeI2C

功能	配置作为 I2C 外设的引脚
函数原型	void GPIOPinTypeI2C（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit－packed）
描述	只有正确配置 I2C 引脚才能使该功能正常。此函数为这些引脚提供了正确的配置
返回参数	无

表 C-31　GPIOPinTypeI2CSCL

功能	配置作为 I2C 外设 SCL 信号的引脚
函数原型	void GPIOPinTypeI2CSCL（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit－packed）
描述	只有正确配置 I2C 引脚才能使功能正常。此函数为 SCL 引脚提供了正确的配置
返回参数	无

表 C-32　GPIOPinTypeKBColumn

功能	配置作为扫描矩阵键盘列输入的引脚
函数原型	void　GPIOPinTypeKBColumn（uint32_t ui32Port, uint8_t ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit－packed）
描述	只有正确配置扫描矩阵键盘输入引脚才能使功能正常，此函数为这些引脚提供了正确的配置
返回参数	无

表 C-33　GPIOPinTypeKBRow

功能	配置作为扫描矩阵键盘行入的引脚
函数原型	void　GPIOPinTypeKBRow（uint32_t ui32Port, uint8_t ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit－packed）
描述	只有正确配置扫描矩阵键盘输出引脚才能使功能正常，此函数为这些引脚提供了正确的配置
返回参数	无

表 C-34　GPIOPinTypeLCD

功能	配置作为 LCD 控制器的引脚
函数原型	void　GPIOPinTypeLCD（uint32_t ui32Port, uint8_t ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit－packed）
描述	只有正确配置 LCD 控制器引脚才能使功能正常，此函数为这些引脚提供了一个典型的配置；其他配置可能取决于板上的设置
返回参数	无

表 C-35　GPIOPinTypeLEDSeq

功能	配置作为 LED 序列发生器的引脚
函数原型	void　GPIOPinTypeLEDSeq（uint32_t ui32Port, uint8_t ui8Pins）

（续）

参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit - packed）
描述	只有正确配置 LED 序列发生器引脚才能使功能正常，此函数为这些引脚提供了正确的配置
返回参数	无

表 C-36　GPIOPinTypeLPC

功能	配置作为 LPC 模块的引脚
函数原型	void GPIOPinTypeLPC（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit - packed）
描述	只有正确配置 LPC 引脚才能是 LPC 模块功能正常。此函数为这些引脚提供了一种典型的配置；其他配置可能取决于板上的设置
返回参数	无

表 C-37　GPIOPinTypePECIRx

功能	配置用于 PECI 模块接收的引脚
函数原型	void GPIOPinTypePECIRx（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit - packed）
描述	只有正确配置 PECI 接收引脚才能使该功能正常。此函数为该引脚提供了一个典型的配置
返回参数	无

表 C-38　GPIOPinTypePECITx

功能	配置用于 PECI 模块发送的引脚
函数原型	void GPIOPinTypePECITx（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit - packed）
描述	只有正确配置 PECI 发送引脚才能使该功能正常。此函数为该引脚提供了一个典型的配置
返回参数	无

表 C-39　GPIOPinTypePWM

功能	配置作为 PWM 外设的引脚
函数原型	void GPIOPinTypePWM（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit - packed）

（续）

描述	只有正确配置 PWM 引脚才能使该功能正常。此函数为这些引脚提供了典型的配置。其他配置可能取决于板上的设置
返回参数	无

表 C-40　GPIOPinTypeQEI

功能	配置作为 QEI 外设的引脚
函数原型	void GPIOPinTypeQEI（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit – packed）
描述	只有正确配置 QEI 引脚才能使该功能正常。此函数为这些引脚提供了典型的配置。其他配置可能取决于板上的设置
返回参数	无

表 C-41　GPIOPinTypeSSI

功能	配置作为 SSI 外设的引脚
函数原型	void GPIOPinTypeSSI（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit – packed）
描述	只有正确配置 SSI 引脚才能使该功能正常。此函数为这些引脚提供了典型的配置。其他配置可能取决于板上的设置
返回参数	无

表 C-42　GPIOPinTypeTimer

功能	配置作为定时器外设的引脚
函数原型	void GPIOPinTypeTimer（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit – packed）
描述	只有正确配置 CCP 引脚才能使定时器外设功能正常。此函数为这些引脚提供了典型的配置。其他配置可能取决于板上的设置（例如，使用片上上拉）
返回参数	无

表 C-43　GPIOPinTypeUART

功能	配置作为 UART 外设的引脚
函数原型	void GPIOPinTypeUART（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit – packed）
描述	只有正确配置 UART 引脚才能使该功能正常。此函数为这些引脚提供了典型的配置。其他配置可能取决于板上的设置
返回参数	无

表 C-44　GPIOPinTypeUSBAnalog

功能	配置作为 USB 外设的引脚
函数原型	void GPIOPinTypeUSBAnalog（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit－packed）
描述	只有正确配置某些 USB 模拟引脚才能使该功能正常。此函数为任何 USB 引脚提供了正确的配置。这也可以用来配置 EPEN 和 PFAULT 引脚，使之不再使用 USB 控制器
返回参数	无

表 C-45　GPIOPinTypeUSBDigital

功能	配置作为 USB 外设的引脚
函数原型	void GPIOPinTypeUSBDigital（uint32_t　ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit－packed）
描述	只有正确配置某些 USB 数字引脚才能使该功能正常。此函数为数字 USB 引脚提供了典型的配置。其他配置可能取决于板上的设置 此函数应该只用于 EPEN 和 PFAULT 引脚。因为其他 USB 引脚本质上都是模拟引脚，或者其他引脚在没有 OTG 功能时是不能被设备使用的
返回参数	无

表 C-46　GPIOPinTypeWakeHigh

功能	配置作为高电平唤醒休眠源的引脚
函数原型	void　GPIOPinTypeWakeHigh（uint32_t ui32Port, uint8_t ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit－packed）
描述	只有正确配置高电平唤醒休眠源引脚才能使该功能正常。此函数为这些引脚提供了正确的配置
返回参数	无

表 C-47　GPIOPinTypeWakeLow

功能	配置作为低电平唤醒休眠源的引脚
函数原型	void　GPIOPinTypeWakeLow（uint32_t ui32Port, uint8_t　ui8Pins）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit－packed）
描述	只有正确配置低电平唤醒休眠源引脚才能使该功能正常。此函数为这些引脚提供了正确的配置
返回参数	无

表 C-48　GPIOPinWakeStatus

功能	获取唤醒引脚的状态
函数原型	uint32_t　GPIOPinWakeStatus（uint32_t ui32Port）
参数	ui32Port　GPIO 端口的基地址

（续）

描述	该函数返回 GPIO 唤醒引脚状态的值。返回的位字段通过 0 或 1 显示引脚状态的低或高
返回参数	返回唤醒引脚的状态

表 C-49　GPIOPinWrite

功能	写入一个值到指定引脚
函数原型	void GPIOPinWrite（uint32_t　ui32Port, uint8_t　ui8Pins, uint8_t　ui8Val）
参数	ui32Port　GPIO 端口的基地址 ui8Pins　引脚的位填充（bit - packed） ui8Val　写入到引脚的值
描述	写入到由 ui8Pins 指定的输出引脚相应位的值。而写入到配置为输入引脚的值无影响
返回参数	无



附录 D　第 6 章附录：模拟比较器固件库函数简介

本章附录将简要介绍模拟比较器固件库函数的参数定义及功能说明。

表 D-1　ComparatorConfigure

作用	配置模拟器
函数原型	void　ComparatorConfigure（uint32_t ui32Base, uint32_t ui32Comp, uint32_t ui32Config）
参数	ui32Base　比较器模块的基地址 ui32Comp　比较器的索引 ui32Config　比较器的配置
描述	该函数配置模拟比较器。参数 ui32Config 是 COMP_TRIG_xxx、COMP_INT_xxx、COMP_ASRCP_xxx 和 COMP_OUTPUT_xxx 值的逻辑或运算 ① COMP_TRIG_xxx 取值如下： ➢ COMP_TRIG_NONE　//不触发 ADC ➢ COMP_TRIG_HIGH　//当比较器输出为高电平时触发 ADC ➢ COMP_TRIG_LOW　//当比较器输出为低电平时触发 ADC ➢ COMP_TRIG_FALL　//当比较器输出下降沿时触发 ADC ➢ COMP_TRIG_RISE　//当比较器输出上升沿时触发 ADC ➢ COMP_TRIG_BOTH　//当比较器输出双边沿时触发 ADC ② COMP_INT_xxx 取值如下： ➢ COMP_INT_HIGH　//当比较器输出高电平时产生中断 ➢ COMP_INT_LOW　//当比较器输出低电平时产生中断 ➢ COMP_INT_FALL　//当比较器输出下降沿时产生中断 ➢ COMP_INT_RISE　//当比较器输出上升沿时产生中断 ➢ COMP_INT_BOTH　//当比较器输出双边沿时产生中断
描述	③ COMP_ASRCP_xxx 取值如下： ➢ COMP_ASRCP_PIN　//使用专用的 Comp + 引脚作为基准电压 ➢ COMP_ASRCP_PIN0　//使用专用的 Comp0 + 引脚作为基准电压 ➢ COMP_ASRCP_REF　//使用内部产生的电压作为基准电压 ④ COMP_OUTPUT_xxx 取值如下： ➢ COMP_OUTPUT_NORMAL //使能比较器非反相输出端（同相端） ➢ COMP_OUTPUT_INVERT　//使能比较器的反相输出端
返回值	无

表 D-2　ComparatorIntClear

作用	清除比较器中断
函数原型	void　ComparatorIntClear（uint32_t ui32Base, uint32_t ui32Comp）
参数	ui32Base　比较器模块的基地址 ui32Comp　比较器的索引
描述	清除比较器中断，使之不再有效。该函数必须在中断处理程序中被调用，以防止处理器退出时立即再次被调用。注意：对于电平触发的中断，在中断无效前不能将它清除
返回值	无

表 D-3　ComparatorIntDisable

作用	禁用比较器中断
函数原型	void　ComparatorIntDisable（uint32_t ui32Base, uint32_t ui32Comp）
参数	ui32Base　比较器模块的基地址 ui32Comp　比较器的索引
描述	该函数禁止指定比较器产生中断
返回值	无

表 D-4　ComparatorIntEnable

作用	使能比较器中断
函数原型	void　ComparatorIntEnable（uint32_t ui32Base, uint32_t ui32Comp）
参数	ui32Base　比较器模块的基地址 ui32Comp　比较器的索引
描述	该函数使能指定比较器产生中断
返回值	无

表 D-5　ComparatorIntRegister

作用	注册比较器中断的中断处理程序
函数原型	void　ComparatorIntRegister（uint32_t ui32Base, uint32_t ui32Comp, void（ * pfnHandler）（void））
参数	ui32Base　比较器模块的基地址 ui32Comp　比较器的索引 pfnHandler　指向当比较器产生中断时被调用函数的指针
描述	该函数在比较器中断发生时用来设置被调用的处理程序，并且使能中断控制器中的中断。中断处理程序通过调用 ComparatorIntClear( ) 函数来负责清楚中断源
返回值	无

表 D-6　ComparatorIntStatus

作用	获取当前的中断状态
函数原型	bool　ComparatorIntStatus（uint32_t ui32Base, uint32_t ui32Comp, bool bMasked）

（续）

参数	ui32Base　比较器模块的基地址 ui32Comp　比较器的索引 bMasked　若需要原始中断状态，bMasked 为 false；若需要屏蔽中断，bMasked 为 true
描述	该函数返回比较器的中断状态。要么返回原始中断状态；要么返回屏蔽中断状态
返回值	如果中断有效，则返回 true；否则返回 false

表 D-7　ComparatorIntUnregister

作用	注销比较器中断的中断处理程序
函数原型	void　ComparatorIntUnregister（uint32_t ui32Base， uint32_t ui32Comp）
参数	ui32Base　比较器模块的基地址 ui32Comp　比较器的索引
描述	在比较器中断产生时，该函数用于清除被调用的处理程序。该函数也会屏蔽中断控制器中的中断，使中断处理程序就不会被再次调用
返回值	无

表 D-8　ComparatorRefSet

作用	设置比较器内部的基准电压
函数原型	Void　ComparatorRefSet（uint32_t ui32Base， uint32_t ui32Ref）
参数	ui32Base　比较器模块的基地址 ui32Ref　基准电压值
描述	该函数设置内部的基准电压值。电压值可指定为如下值之一： ➢ COMP_REF_OFF　　//关闭参考电压 ➢ COMP_REF_0V　　//设置参考电压为 0V ➢ COMP_REF_0_1375V ➢ COMP_REF_0_275V ➢ COMP_REF_0_4125V ➢ COMP_REF_0_55V ➢ COMP_REF_0_6875V ➢ COMP_REF_0_825V ➢ COMP_REF_0_928125V ➢ COMP_REF_0_9625V ➢ COMP_REF_1_03125V t ➢ COMP_REF_1_134375V ➢ COMP_REF_1_1V ➢ COMP_REF_1_2375V ➢ COMP_REF_1_340625V ➢ COMP_REF_1_375V ➢ COMP_REF_1_44375V ➢ COMP_REF_1_5125V ➢ COMP_REF_1_546875V ➢ COMP_REF_1_65V ➢ COMP_REF_1_753125V ➢ COMP_REF_1_7875V ➢ COMP_REF_1_85625V ➢ COMP_REF_1_925V ➢ COMP_REF_1_959375V ➢ COMP_REF_2_0625V ➢ COMP_REF_2_165625V ➢ COMP_REF_2_26875V ➢ COMP_REF_2_371875V　　//设置基准电压为 2. 371875 V
返回值	无

表 D-9　ComparatorValueGet

作用	获取当前比较器的输出值
函数原型	bool　ComparatorValueGet（uint32_t ui32Base, uint32_t ui32Comp）
参数	ui32Base　比较器模块的基地址 ui32Comp　比较器的索引
描述	该函数获取当前比较器的输出值
返回值	若输出值为高电平，则返回为 true，若输出值为低电平，则返回 false



附录 E　第 7 章附录：SysTick 与 NVIC 固件库函数简介

本章附录将介绍 SysTick 与 NVIC 的固件库函数。

E. 1　SysTick 固件库函数

表 E-1　SysTickDisable

功能	禁止 SysTick 计数器
函数原型	void　SysTickDisable（void）
描述	该函数禁止系统定时器计数器。若已注册了一个中断处理程序，那么在系统定时器重启前，将不会被调用该中断处理程序
返回参数	无

表 E-2　SysTickEnable

功能	使能 SysTick 计数器
函数原型	void　SysTickEnable（void）
描述	该函数启动系统定时器计数器。若已注册了一个中断处理程序，那么在系统定时器计数器翻转时，将调用该中断处理程序
返回参数	无

表 E-3　SysTickIntDisable

功能	禁止 SysTick 中断
函数原型	void　SysTickIntDisable（void）
描述	该函数禁止系统定时器中断，防止其被反射到处理器中
返回参数	无

表 E-4　SysTickIntEnable

功能	使能 SysTick 中断
函数原型	void　SysTickIntEnable（void）
描述	该函数使能系统定时器中断，允许其反射到处理器中
返回参数	无

表 E-5　SysTickIntRegister

功能	注册 SysTick 中断的中断处理程序
函数原型	void　SysTickIntRegister（void（＊pfnHandler）（void））
参数	pfnHandler　指向系统定时器中断发生时被调函数的指针
描述	该函数注册在系统定时器中断发生时被调用的中断处理程序
返回参数	无

表 E-6　SysTickIntUnregister

功能	注销 SysTick 中断的中断处理程序
函数原型	void　SysTickIntUnregister（void）
描述	该函数注销在系统定时器中断发生时被调用的中断处理程序
返回参数	无

表 E-7　SysTickPeriodGet

功能	获取 SysTick 计数器的周期
函数原型	uint32_t　SysTickPeriodGet（void）
描述	该函数返回 SysTick 计数器缠绕（wrap）计数的速度，它等于两个中断之间处理器的时钟数
返回参数	返回 SysTick 计数器的周期

表 E-8　SysTickPeriodSet

功能	设置 SysTick 计数器的周期
函数原型	void　SysTickPeriodSet（uint32_t ui32Period）
参数	ui32Period　每个 SysTick 计数器周期的时钟滴答数（1≤SysTick≤16，777，216,）
描述	该函数设置 SysTick 计数器缠绕（wrap）计数的速度，它等于两个中断之间处理器的时钟数
返回参数	无

表 E-9　SysTickValueGet

功能	获取 SysTick 计数器的当前值
函数原型	uint32_t　SysTickValueGet（void）
描述	该函数返回 SysTick 计数器的当前值（－1≤该值≤0）
返回参数	返回 SysTick 计数器的当前值

E.2　NVIC 固件库函数

下面将介绍 NVIC 固件库函数的参数定义及功能说明。

表 E-10　IntDisable

功能	禁止中断
函数原型	void IntDisable（uint 32_t ui32Interrupt）
参数	ui32Interrupt　禁止指定的中断

（续）

描述	禁止中断控制器中的指定中断。其他使能的中断不会受到该函数的影响
返回参数	无

表 E-11　IntEnable

功能	使能中断
函数原型	void　IntEnable（uint32_t ui32Interrupt）
输入参数	ui32Interrupt　禁止指定的中断
描述	使能中断控制器中的指定中断
返回参数	无

表 E-12　IntIsEnabled

功能	返回外设中断是否使能
函数原型	uint32_t　IntIsEnabled（uint32_t ui32Interrupt）
输入参数	ui32Interrupt　指定检查中断
描述	该函数检查中断控制器中指定的中断是否使能
返回参数	若中断使能将返回一个非零值

表 E-13　IntMasterDisable

功能	禁用处理器中断
函数原型	bool IntMasterDisable（void）
描述	该函数防止处理器收到中断。此函数不影响在中断控制器中的中断使能设置；它只管控从控制器到处理器的单一中断
返回参数	若函数在调用时中断已被禁止，则返回 true；若中断已被使能，则返回 false

表 E-14　IntMasterEnable

功能	使能处理器的中断
函数原型	bool IntMasterEnable（void）
描述	该函数允许处理器响应中断。此函数不影响在中断控制器中的中断使能设置；它只管控从控制器到处理器的单一中断
返回参数	若函数在调用时中断已被禁止，则返回 true；若中断已被使能，则返回 false

表 E-15　IntPendClear

功能	未被挂起的中断
函数原型	void　IntPendClear（uint32_t ui32Interrupt）
输入参数	ui32Interrupt　指定的中断未被挂起
描述	指定在中断控制器中未被挂起的中断。这将导致任何先前产生的没有被处理的中断（由于更高优先级中断或中断尚未被使能）将被丢弃
返回参数	无

表 E-16　IntPendSet

功能	挂起的中断
函数原型	void IntPendSet（uint32_t ui32Interrupt）
输入参数	ui32Interrupt　指定挂起的中断
描述	指定中断控制器中被挂起的中断。挂起中断可使中断控制器根据当前的中断状态的优先级，在下一个可用的时间去执行相应的中断处理程序
返回参数	无

表 E-17　IntPriorityGet

功能	获取中断的优先级
函数原型	int32_t ntPriorityGet（uint32_t ui32Interrupt）
输入参数	ui32Interrupt　指定讨论的中断
描述	该函数获取中断的优先级。参考 IntPrioritySet（）函数对优先级值的定义
返回参数	返回中断的优先级，若指定一个无效的中断将返回 -1

表 E-18　IntPriorityGroupingGet

功能	获取中断控制器中的优先级分组
函数原型	uint32_t　IntPriorityGroupingGet（void）
描述	该函数返回中断优先级规范中抢占式优先级与子优先级等级之间的拆分
返回参数	抢占式优先级的位数

表 E-19　IntPriorityGroupingSet

功能	设置中断控制器的优先级分组
函数原型	void IntPriorityGroupingSet（uint32_t ui32Bits）
输入参数	ui32Bits　指定可抢占的优先级的位数
描述	这个函数指定中断优先级规范中抢占式优先级与子优先级等级之间的拆分。分组值的范围依赖于硬件决定；对于 TIVA C 和 E 系列家族，可用 3 位来决定硬件中断优先级，因此优先级分组值 3 ~ 7 有同样的效果
返回参数	无

表 E-20　IntPriorityMaskGet

功能	获取优先级的屏蔽等级
函数原型	uint32_t　IntPriorityMaskGet（void）
描述	这个函数获取当前屏蔽中断优先级等级的设置。返回值是所有中断的优先级，对于优先级较低的中断将被屏蔽。较小的数字对应较高的中断优先级，值为 0 时将禁止优先级屏蔽。而对于硬件优先级机制只需查看优先级上面的 N 位（其中 N = 3 是 TIVA C 和 E 系列家族），所以任何优先级排序都必须在这些位中进行
返回参数	返回中断优先级屏蔽的值

表 E-21　IntPriorityMaskSet

功能	设置优先级的屏蔽等级
函数原型	void　IntPriorityMaskSet（uint32_t ui32PriorityMask）

（续）

输入参数	ui32PriorityMask　待屏蔽的优先级
描述	此函数设置屏蔽中断优先级，因此，所有指定的中断或优先级较低的中断将被屏蔽。掩蔽中断可用于全局禁止优先级低于预定阈值的一组中断，值为 0 时将禁止优先级屏蔽
返回参数	无

表 E-22　IntPrioritySet

功能	设置中断的优先级
函数原型	void IntPrioritySet（uint32_t ui32Interrupt, uint8_t ui8Priority）
输入参数	ui32Interrupt　指定讨论的中断 ui8Priority　指定的中断优先级
描述	该函数用于设置中断的优先级。当多个中断同时有效时，具有最高优先级的中断比那些优先级低的中断先行处理。数字越小对应的中断优先级越高，优先级为 0 的中断优先级最高 硬件优先级机制只需查看优先级上面的 N 位（其中 N＝3 是 TIVA C 和 E 系列家族），所以任何优先级都必须在这些位中进行。剩余的位可用于中断源的子优先级，也可用于将来的器件中。这样的安排允许优先级可迁移到不同的 NVIC 中而不改变总的中断优先级
返回参数	无

表 E-23　IntRegister

功能	注册发生中断时被调用的函数
函数原型	void IntRegister（uint32_t ui32Interrupt, void（＊pfnHandler）（void））
输入参数	ui32Interrupt　指定讨论的中断 pfnHandler　指向被调用函数的指针
描述	当给定处理器中断有效时，此函数用于指定被调用的处理函数。当中断发生时，如果通过 IntEnable( )已使能中断，则在中断上下文中的处理函数将被调用。因为处理函数可以抢占其他代码，因此必须小心保护处理程序和其他非处理程序代码所访问的存储器或外设
返回参数	无

表 E-24　IntTrigger

功能	触发中断
函数原型	void IntTrigger（uint32_t ui32Interrupt）
输入参数	ui32Interrupt　指定的触发中断
描述	该函数执行中断的软件触发。若中断控制器中相应的中断行为有效，则以相同的方式来处理中断
返回参数	无

表 E-25　IntUnregister

功能	注销发生中断时被调用的函数
函数原型	void IntUnregister（uint32_t ui32Interrupt）
输入参数	ui32Interrupt　指定的讨论中断
描述	当给定处理器中断有效时，此函数用于指出无处理函数被调用。如果必要的话，中断源将通过 IntDisable( )来自动清除
返回参数	无



附录 F　第 8 章附录：I2C 固件库函数简介

本章附录将简要介绍 I2C 固件库的参数定义及函数的功能说明。

表 F-1　I2CFIFODataGet

功能	从 I2C 接收 FIFO 读取一个字节
函数原型	uint32_t　I2CFIFODataGet（uint32_t ui32Base）
参数	ui32Base　I2C 主机或从机模块的基地址
描述	该函数从 I2C 接收 FIFO 读取一个字节的数据，以及由参数 pui8Data 在指定的保存单元。如果没有可用的数据，该函数将一直等待到收到数据才会返回
返回	数据字节

表 F-2　I2CFIFODataGetNonBlocking

功能	从 I2C 接收 FIFO 读取一个字节
函数原型	uint32_t　I2CFIFODataGetNonBlocking（uint32_t　ui32Base， uint8_t ＊pui8Data）
参数	ui32Base　I2C 主机或从机模块的基地址 pui8Data　指向待读取数据的保存单元的指针
描述	该函数从 I2C 接收 FIFO 读取一个字节的数据，以及由参数 pui8Data 在指定的保存单元。如果没有可用的数据，该函数将返回 0
返回	从 I2C 接收 FIFO 读出单元（element）的数目

表 F-3　I2CFIFODataPut

功能	写一个数据字节到 I2C 发送 FIFO
函数原型	void　I2CFIFODataPut（uint32_t ui32Base，uint8_t ui8Data）
参数	ui32Base　I2C 主机或从机模块的基地址 pui8Data　保存在发送 FIFO 中的数据
描述	该函数将据添一加个字节数据到 I2C 发送 FIFO 中。如果在 FIFO 中没有可用的空间，这个函数将等待到有可用的空间才会返回
返回	无

表 F-4　I2CFIFODataPutNonBlocking

功能	为 I2C 模块注册一个中断处理程序
	uint32_t　I2CFIFODataPutNonBlocking（uint32_t ui32Base，uint8_t ui8Data）
参数	ui32Base　I2C 主机或从机模块的基地址 pui8Data　保存在发送 FIFO 中的数据
描述	该函数将据添一加个字节数据到 I2C 发送 FIFO 中。如果在 FIFO 中没有可用的空间，该函数将返回 0
返回	无

表 F-5　I2CFIFOStatus

功能	获取当前的 FIFO 状态
函数原型	uint32_t　I2CFIFOStatus（uint32_t ui32Base）
参数	ui32Base　I2C 主机或从机模块的基地址

（续）

描述	此函数获取发送（TX）和接收（RX）FIFO 的状态。可使用 I2CTxFIFOConfigSet( ) 函数来设置发送 FIFO 的触发深度与使用 I2CTxFIFOConfigSet( ) 函数来设置接收 F IFO 的触发深度
返回	返回 FIFO 的状态，枚举为包含下列值的位字段： ➢ I2C_FIFO_RX_BEL ➢ OW_TRIG_LEVEL ➢ I2C_FIFO_RX_FULL ➢ I2C_FIFO_RX_EMPTY ➢ I2C_FIFO_TX_BELOW_TRIG_LEVEL ➢ I2C_FIFO_TX_FULL ➢ I2C_FIFO_TX_EMPTY

表 F-6　I2CintRegister

功能	注册 I2C 模块的中断处理程序
函数原型	void I2CIntRegister（uint32_t ui32Base，void（ * pfnHandler）（void））
参数	ui32Base　I2C 主机模块的基地址 pfnHandler　指向当 I2C 中断发生时待调用函数的指针
描述	当 I2C 中断发现时该函数设置待调用的处理程序。这个函数使能中断控制器中的全局中断；具体的 I2C 中断必须通过 I2CMasterIntEnable( ) 和 I2CSlaveIntEnable( ) 函数使能。若有必要，中断处理程序可通过 I2CMasterIntClear( ) 和 I2CSlaveIntClear( ) 函数来负责清除中断源
返回	无

表 F-7　I2CintUnregister

功能	注销 I2C 模块的中断处理程序
函数原型	void　I2CIntUnregister（uint32_t　ui32Base）
参数	ui32Base　I2C 主机模块的基地址
描述	当 I2C 发生中断时该函数清除待调用的处理程序。这个函数也可屏蔽掉中断控制器中的中断，以使中断处理程序不再被调用
返回	无

表 F-8　I2CMasterBurstCountGet

功能	返回突发传送计数器的的当前值
函数原型	uint32_t　I2CMasterBurstCountGet（uint32_t ui32Base）
参数	ui32Base　I2C 主机模块的基地址
描述	该函数返回用于 FIFO 机制突发传送计数器的当前值。软件可以使用该值来确定在传输中保持多少个字节，或者，若发生了错误时，在哪里进行突发传输操作
返回	无

表 F-9　I2CMasterBurstLengthSet

功能	指示 I2C 总线是否忙碌
函数原型	void　I2CMasterBurstLengthSet（uint32_t ui32Base，uint8_t ui8Length）
参数	ui32Base　I2C 主机模块的基地址 ui8Length　突发传输的长度
描述	该函数配置 I2C 主机 FIFO 操作的突发传输长度。突发字段被限定于 8 位或 256 字节之内。突发长度适用于单一的 I2CMCS BURST 操作，这意味着指定的突发长度仅为当前操作(可以发送或接收)。每次进行突发操作时，必须使用 I2CMasterControl( ) 写 I2CMCS 中的 BURST 位来配置突发长度的优先级
返回	无

表 F-10　I2CMasterBusBusy

功能	指示 I2C 总线是否忙碌
函数原型	bool　I2CMasterBusBusy（uint32_t　ui32Base）
参数	ui32Base　I2C 主机模块的基地址
描述	该函数返回 I2C 总线是否忙碌的指示。这个函数可以用于多主机环境中以确定当前是否有另一主机正在使用总线
返回	若 I2C 总线处于忙碌状态，则返回 ture；否则，返回 false

表 F-11　I2CMasterBusy

功能	指示 I2C 主机是否忙碌
函数原型	tBoolean　I2CMasterBusy（unsigned long ulBase）
参数	ui32Base　I2C 主机模块的基地址
描述	该函数返回 I2C 主机是否正忙于发送或接收数据的指示
返回	若 I2C 总线处于忙碌状态，则返回 ture；否则，返回 false

表 F-12　I2CMasterControl

功能	控制 I2C 主机模块的状态
函数原型	void I2CMasterControl（uint32_t　ui32Base，uint32_t　ui32Cmd）
参数	ui32BaseI2C　主机模块的基地址 ui32Cmd　下达给 I2C 主机模块的命令
描述	该函数用于控制主机模块发送和接收操作的状态。 参数 ui32Cmd 为下列值之一： ➢ I2C_MASTER_CMD_SINGLE_SEND ➢ I2C_MASTER_CMD_SINGLE_RECEIVE ➢ I2C_MASTER_CMD_BURST_SEND_START ➢ I2C_MASTER_CMD_BURST_SEND_CONT ➢ I2C_MASTER_CMD_BURST_SEND_FINISH ➢ I2C_MASTER_CMD_BURST_SEND_ERROR_STOP ➢ I2C_MASTER_CMD_BURST_RECEIVE_START ➢ I2C_MASTER_CMD_BURST_RECEIVE_CONT ➢ I2C_MASTER_CMD_BURST_RECEIVE_FINISH ➢ I2C_MASTER_CMD_BURST_RECEIVE_ERROR_STOP ➢ I2C_MASTER_CMD_QUICK_COMMAND ➢ I2C_MASTER_CMD_HS_MASTER_CODE_SEND ➢ I2C_MASTER_CMD_FIFO_SINGLE_SEND
描述	➢ I2C_MASTER_CMD_FIFO_SINGLE_RECEIVE ➢ I2C_MASTER_CMD_FIFO_BURST_SEND_START ➢ I2C_MASTER_CMD_FIFO_BURST_SEND_CONT ➢ I2C_MASTER_CMD_FIFO_BURST_SEND_FINISH ➢ I2C_MASTER_CMD_FIFO_BURST_SEND_ERROR_STOP ➢ I2C_MASTER_CMD_FIFO_BURST_RECEIVE_START ➢ I2C_MASTER_CMD_FIFO_BURST_RECEIVE_CONT ➢ I2C_MASTER_CMD_FIFO_BURST_RECEIVE_FINISH ➢ I2C_MASTER_CMD_FIFO_BURST_RECEIVE_ERROR_STOP
返回	无

表 F-13　I2CMasterDataGet

功能	接收一个发送给 I2C 主机的字节
函数原型	uint32_t　I2CMasterDataGet（uint32_t　ui32Base）
参数	ui32Base　I2C 主机模块的基地址

（续）

描述	该函数读取 I2C 主机数据寄存器中一个字节的数据
返回	返回从 I2C 主机接收的字节并强制转换为 uint32_t 类型

表 F-14　I2CMasterDataPut

功能	从 I2C 主机发送一个字节
函数原型	void I2CMasterDataPut（uint32_t ui32Base, uint8_t ui8Data）
参数	ui32Base　I2C 主机模块的基地址 ui8Data　从 I2C 主机被发送的数据
描述	该函数把提供的数据保存到 I2C 主机数据寄存器中
返回	无

表 F-15　I2CMasterDisable

功能	禁止 I2C 主机模块
函数原型	void I2CMasterDisable（uint32_t　ui32Base）
参数	ui32Base　I2C 主机模块的基地址
描述	该函数禁止 I2C 主机块的操作
返回	无

表 F-16　I2CMasterEnable

功能	使能 I2C 主机模块
函数原型	void I2CMasterEnable（uint32_t ui32Base）
参数	ui32Base　I2C 主机模块的基地址
描述	该函数使能 I2C 主机块操作
返回	无

表 F-17　I2CMasterErr

功能	获取 I2C 主机模块的错误状态
函数原型	uint32_t　I2CMasterErr（uint32_t ui32Base）
参数	ui32Base　I2C 主机模块的基地址
描述	该函数用于获取主机模块发送和接收操作的错误状态
返回	返回如下之一的错误状态： ➢ I2C_MASTER_ERR_NONE ➢ I2C_MASTER_ERR_ADDR_ACK ➢ I2C_MASTER_ERR_DATA_ACK ➢ I2C_MASTER_ERR_ARB_LOST

表 F-18　I2CMasterGlitchFilterConfigSet

功能	配置 I2C 主机故障过滤器
函数原型	void　I2CMasterGlitchFilterConfigSet（uint32_t ui32Base, uint32_t ui32Config）
参数	ui32Base　I2C 主机模块的基地址 ui32Config　故障过滤器配置

（续）

描述	该函数配置 I2C 主机故障过滤器。通过对 ui32config 价值决定故障滤波器的采样范围，值传递给 ui32Config 以确定故障滤波器的采样范围，它可配置为 1 ~32 个系统 clock cycles. 时钟周期。故障滤波器的默认配置为 0 个系统时钟周期，即故障滤波器被禁止 ui32config 字段应为下列的任意值： ➢ I2C_MASTER_GLITCH_FILTER_DISABLED ➢ I2C_MASTER_GLITCH_FILTER_1 ➢ I2C_MASTER_GLITCH_FILTER_2 ➢ I2C_MASTER_GLITCH_FILTER_3 ➢ I2C_MASTER_GLITCH_FILTER_4 ➢ I2C_MASTER_GLITCH_FILTER_8 ➢ I2C_MASTER_GLITCH_FILTER_16 ➢ I2C_MASTER_GLITCH_FILTER_32
返回	无

表 F-19　I2CMasterInitExpClk

功能	初始化 I2C 主机模块
函数原型	void　I2CMasterInitExpClk（uint32_t ui32Base，uint32_t ui32I2CClk，bool bFast）
参数	ui32Base　I2C 主机模块的基地址 ui32I2CClk　提供给 I2C 模块的时钟速率 bFast　设置快速数据传输
描述	该函数通过设置主机的总线速度和使能 I2C 主机模块来初始化 I2C 主机模块的操作 如果参数 bFast 为 true，则主机模块以 400Kbps 传输数据；否则将以 100Kbps 传输数据。若需要快速 + 模式（1 Mbps），软件应在调用这个函数前手工编写 I2CMTPR。对于高速（3.4Mbps）模式，在以 100Kbps 或 400Kbps 的速率与从机进行最初的通信之后，需用一个特定的命令来将其切换到更快的时钟速率 外设时钟与处理器时钟相同。该值由 SysCtlClockGet( ) 返回，或者，在该时钟为一个已知值或常量时，可表示成显式地硬编码
返回	无

表 F-20　I2CMasterIntClear

功能	清除 I2C 主机的中断源
函数原型	void I2CMasterIntClear（uint32_t　ui32Base）
参数	ui32Base　I2C 主机模块的基地址
描述	清除 I2C 主机的中断源，使之不再有效。该函数必须在中断处理程序中被调用，防止在退出时中断立即被再次触发
返回	无

表 F-21　I2CMasterIntClearEx

功能	清除 I2C 主机的中断源
函数原型	void I2CMasterIntClearEx（uint32_t　ui32Base，uint32_t ui32IntFlags）
参数	ui32Base　I2C 主机模块的基地址 ui32IntFlags　待清除中断源的屏蔽位
描述	指定 I2C 主机要清除的中断源，使之不再有效。这个函数必须在中断处理程序中被调用，以防止在退出时中断立即被再次触发
返回	无

表 F-22　I2CMasterIntDisable

功能	禁止 I2C 主机中断
函数原型	void　I2CMasterIntDisable（uint32_t ui32Base）
参数	ui32Base　I2C 主机模块的基地址
描述	该函数清除 I2C 主机的中断源
返回	无

表 F-23　I2CMasterIntDisableEx

功能	清除单个 I2C 主机的中断源
函数原型	void　I2CMasterIntDisableEx（uint32_t　ui32Base，uint32_t　ui32IntFlags）
参数	ui32Base　I2C 主机模块的基地址 ui32IntFlags　待清除中断源的屏蔽位
描述	这个函数关闭指定的 I2C 主机中断源。只有使能这个中断源才可以反映到处理器中断；关闭中断源不会对处理器造成影响
返回	无

表 F-24　I2CMasterIntEnable

功能	使能 I2C 主机中断
函数原型	void I2CMasterIntEnable（uint32_t ui32Base）
参数	ui32Base　I2C 主机模块的基地址
描述	该函数使能 I2C 主机的中断源
返回	无

表 F-25　I2CmasterIntEnableEx

功能	使能单个 I2C 主机的中断源
函数原型	void I2CMasterIntEnableEx（uint32_t　ui32Base， uint32_t　ui32IntFlags）
参数	ui32Base　I2C 主机模块的基地址 ui32IntFlags　待使能中断源的屏蔽位
描述	该函数使能指定 I2C 主机的中断源。只有使能这个中断源才可以反映到处理器中断；关闭中断源不会对处理器造成影响 参数 uiIntFlags 是以下任意值的逻辑或 ➢ I2C_MASTER_INT_RX_FIFO_FULL//RX FIFO 满中断 ➢ I2C_MASTER_INT_TX_FIFO_EMPTY //TX FIFO 空中断 ➢ I2C_MASTER_INT_RX_FIFO_REQ// RX FIFO 请求中断 ➢ I2C_MASTER_INT_TX_FIFO_REQ// TX FIFO 请求中断 ➢ I2C_MASTER_INT_ARB_LOST//仲裁丢失中断 ➢ I2C_MASTER_INT_STOP //停止条件中断 ➢ I2C_MASTER_INT_START //开始条件中断 ➢ I2C_MASTER_INT_NACK //地址/数据不应答（NACK）中断 ➢ I2C_MASTER_INT_TX_DMA_DONE//TX DMA 完成中断 ➢ I2C_MASTER_INT_RX_DMA_DONE//RX DMA 完成中断 ➢ I2C_MASTER_INT_TIMEOUT //时钟超时中断 ➢ I2C_MASTER_INT_DATA //数据中断
返回	无

表 F-26　I2CMasterIntStatus

功能	获取当前 I2C 主机的中断状态
函数原型	bool I2CMasterIntStatus（uint32_t ui32Base，bool bMasked）
参数	ui32Base　I2C 主机模块的基地址 bMasked　若要求原始中断状态，则 bMasked 为 false；若需要屏蔽中断状态，则 bMasked 为 true
描述	该函数返回 I2C 主机模块的中断状态。可返回原始中断状态或允许反映到处理器中的中断状态
返回	返回当前的中断状态，激活返回 true，未激活则返回 false

表 F-27　I2CMasterIntStatusEx

功能	获取当前 I2C 主机的中断状态
函数原型	uint32_t I2CMasterIntStatusEx（uint32_t　ui32Base，bool bMasked）
参数	ui32Base　I2C 主机模块的基地址 bMasked　若要求原始中断状态，则 bMasked 为 false；若需要屏蔽中断状态，则 bMasked 为 true
描述	该函数返回 I2C 主机模块的中断状态。可返回原始中断状态或允许反映到处理器中的中断状态
返回	返回当前中断状态，在 I2CMasterIntEnableEx( )中枚举为位字段的值

表 F-28　I2CMasterLineStateGet

功能	读取 SDA 和 SCL 引脚的状态
函数原型	uint32_t I2CMasterLineStateGet（uint32_t ui32Base）
参数	ui32Base　I2C 主机模块的基地址
描述	该函数通过提供的 SDA 和 SCL 引脚的实时值返回 I2C 总线的状态
返回	返回总线的状态，使 SDA 在位位置 1 以及 SCL 在位位置 0（with SDA in bit position 1 and SCL in bit position 0）

表 F-29　I2CMasterSlaveAddrSet

功能	设置 I2C 主机在总线上的地址
函数原型	void I2CMasterSlaveAddrSet（uint32_t ui32Base，uint8_t ui8SlaveAddr，bool bReceive）
参数	ui32Base　I2C 主机模块的基地址 ui8SlaveAddr　7 位从机地址 bReceive　指示与从机通信的类型标志
描述	当启动通信时，该函数设置放置到总线上的 I2C 主机地址。当参数 bReceive 设置成 true 时，地址将指示 I2C 主机正启动从机读；否则指示 I2C 主机正启动从机写
返回	无

表 F-30　I2CMasterTimeoutSet

功能	设置主时钟的超时值
函数原型	void I2CMasterTimeoutSet（uint32_t ui32Base，uint32_t ui32Value）
参数	ui32Base　I2C 主机模块的基地址 ui32Value　超时的 I2C 时钟数
描述	该函数使能与配置在 I2C 外备的时钟低超时特性。此特性以 12 位的计数器执行，其高 8 位是可编程的
返回	无

表 F-31　I2CRxFIFOConfigSet

功能	配置 I2C 的接收（RX）FIFO
函数原型	void　I2CRxFIFOConfigSet（uint32_t ui32Base，uint32_t ui32Config）
参数	ui32Base　I2C 主机或从机模块的基地址 ui32Config　使用指定宏的配置 FIFO
描述	配置 I2C 外设的接收 FIFO。主机或从机都可以使用接收 FIFO，但它们不能同时被使用。以下宏用来配置带或不带 DMA 的主机或从机的 RX FIFO 行为： ➢ I2C_FIFO_CFG_RX_MASTER ➢ I2C_FIFO_CFG_RX_SLAVE ➢ I2C_FIFO_CFG_RX_MASTER_DMA ➢ I2C_FIFO_CFG_RX_SLAVE_DMA 使用下面其中一个宏来选择 FIFO 的触发深度： ➢ I2C_FIFO_CFG_RX_TRIG_1 ➢ I2C_FIFO_CFG_RX_TRIG_2 ➢ I2C_FIFO_CFG_RX_TRIG_3 ➢ I2C_FIFO_CFG_RX_TRIG_4 ➢ I2C_FIFO_CFG_RX_TRIG_5 ➢ I2C_FIFO_CFG_RX_TRIG_6 ➢ I2C_FIFO_CFG_RX_TRIG_7 ➢ I2C_FIFO_CFG_RX_TRIG_8
返回	无

表 F-32　I2CRxFIFOFlush

功能	刷新 I2C 的接收（RX）FIFO
函数原型	void　I2CRxFIFOFlush（uint32_t ui32Base）
参数	ui32Base　I2C 主机或从机模块的基地址
描述	该函数刷新 I2C 的接收（RX）FIFO
返回	无

表 F-33　I2CSlaveACKOverride

功能	配置 I2C 从机的应答（ACK）重载行为
函数原型	void I2CSlaveACKOverride（uint32_t ui32Base，bool bEnable）
参数	ui32Base　I2C 从机模块的基地址 bEnable　使能或禁止 ACK 重载
描述	该函数使能或禁止 ACK 重载，允许用户程序在 ACK 期间驱动 SDA 上的值
返回	无

表 F-34　I2CSlaveACKValueSet

功能	写应答（ACK）的值
函数原型	void I2CSlaveACKValueSet（uint32_t ui32Base，bool bACK）
参数	ui32Base　I2C 从机模块的基地址 bACK　选择传输中是应答（true）还是不应答（faluse）
描述	该函数在 ACK 周期把所需 ACK 值放置在 SDA 上。写入的值只有当使用 I2CSlaveACKOverride（ ）使能重载 ACK 时才有效
返回	无

表 F-35　I2CSlaveAddressSet

功能	设置 I2C 的从机地址
函数原型	void I2CSlaveAddressSet（uint32_t ui32Base，uint8_t ui8AddrNum，uint8_t ui8SlaveAddr）
参数	ui32Base　I2C 从机模块的基地址 ui8AddrNum　确定设置哪个从机地址 ui8SlaveAddr　7 位从机地址
描述	该函数写数据到指定的从机地址。ulAddrNum 字段指定将配置哪个从机地址
返回	无

表 F-36　I2CSlaveDataGet

功能	接收一个发送给 I2C 从机的字节
函数原型	uint32_t　I2CSlaveDataGet（uint32_t　ui32Base）
参数	ui32Base　I2C 从机模块的基地址
描述	该函数从 I2C 从机数据寄存器中读取一个字节的数据
返回	返回 I2C 从机收到的字节并强制转换成 uint32_t 类型

表 F-37　I2CSlaveDataPut

功能	I2C 从机发送一个字节的数据
函数原型	void I2CSlaveDataPut（uint32_t ui32Base，uint8_t ui8Data）
参数	ui32Base　I2C 从机模块的基地址 ui8Data　I2C 从机待发送的数据
描述	该函数将提供的数据放置到 I2C 从机的数据寄存器中
返回	无

表 F-38　I2CSlaveDisable

功能	禁止 I2C 从机模块
函数原型	voidI2CSlaveDisable（uint32_t ui32Base）
参数	ui32Base　I2C 从机模块的基地址
描述	该函数禁止 I2C 从机模块的操作
返回	无

表 F-39　I2CSlaveEnable

功能	使能 I2C 从机模块
函数原型	void I2CSlaveEnable（uint32_t　ui32Base）
参数	ui32Base　I2C 从机模块的基地址
描述	该函数使能 I2C 从机模块的操作
返回	无

表 F-40　I2CSlaveFIFODisable

功能	禁用 I2C 从机模块使用 FIFO
函数原型	void　I2CSlaveFIFODisable（uint32_t ui32Base）
参数	ui32Base　I2C 从机模块的基地址
描述	该函数禁止 I2C 从机模块使用 FIFO。调用该函数后，虽然 FIFO 被禁止，但从机模块仍然保持有效
返回	无

表 F-41　I2CSlaveFIFOEnable

功能	使能 I2C 从机模块的 FIFO 使用
函数原型	void　I2CSlaveFIFOEnable（uint32_t ui32Base， uint32_t　ui32Config）
参数	ui32Base　I2C 从机模块的基地址 ui32Config　所需的 I2C 从机模块配置
描述	该函数配置 I2C 从机模块的 FIFO 使用。该函数应与 I2CTxFIFOConfigSet( ) 和/或 I2CRxFIFOConfigSet( )组合使用，可配置 FIFO 的触发深度，并告诉 FIFO 硬件是否与 I2C 主机或从机交互。ui32Config 字段为 I2C_SLAVE_TX_FIFO_ENABLE 与 I2C_SLAVE_RX_FIFO_ENABLE 的恰当组合 从机 I2CSCSR 寄存器为只写，那么在调用 I2CSlaveEnable ( )、I2CSlaveDisable 或 I2CSlaveFIFOEnable( )时需重写从机配置。因此，应用软件应先行调用 I2CSlaveEnable ( )，随后是所需 FIFO 配置的 I2CSlaveFIFOEnable ( )
返回	无

表 F-42　I2CSlaveInit

功能	初始化 I2C 的从机模块
函数原型	void I2CSlaveInit（uint32_t ui32Base，uint8_t ui8SlaveAddr）
参数	ui32Base　I2C 从机模块的基地址 ui8SlaveAddr　7 位从机地址
描述	该函数通过配置从机地址与使能 I2C 从机模块来初始化 I2C 从机模块的操作 参数 ui8SlaveAddr 为与 I2C 主机发送的从机地址进行比较的值
返回	无

表 F-43　I2CSlaveIntClear

功能	清除 I2C 从机的中断源
函数原型	void I2CSlaveIntClear（uint32_t　ui32Base）
参数	ui32Base　I2C 从机模块的基地址
描述	清除 I2C 从机的中断源使之不再有效。该函数必须在中断处理程序中被调用，防止在退出时中断立即被再次触发
返回	无

表 F-44　I2CSlaveIntClearEx

功能	清除 I2C 从机的中断源
函数原型	void I2CSlaveIntClearEx（uint32_t ui32Base，uint32_t ui32IntFlags）
参数	ui32Base　I2C 从机模块的基地址 ui32IntFlags　待清除的中断源的屏蔽位
描述	清除指定 I2C 从机的中断源，使之不再有效。该函数必须在中断处理程序中被调用，防止在退出时中断立即被再次触发
返回	无

表 F-45　I2CslaveIntDisable

功能	禁止 I2C 从机的中断
函数原型	void I2CSlaveIntDisable（uint32_t ui32Base）
参数	ui32Base　I2C 从机模块的基地址
描述	该函数禁止 I2C 从机的中断源
返回	无

表 F-46　I2CslaveIntDisableEx

功能	禁止单个 I2C 从机的中断源
函数原型	void I2CSlaveIntDisableEx（uint32_t ui32Base，uint32_t ui32IntFlags）
参数	ui32Base　I2C 从机模块的基地址 ui32IntFlags　待清除中断源的屏蔽位
描述	该函数清除指定 I2C 从机的中断源。只有使能这个中断源才会反映到处理器中断；禁止的中断源不会影响处理器
返回	无

表 F-47　I2CslaveIntEnable

功能	使能 I2C 从机的中断
函数原型	void I2CSlaveIntEnable（uint32_t ui32Base）
参数	ui32Base　I2C 从机模块的基地址
描述	该函数使能 I2C 从机的中断源
返回	无

表 F-48　I2CslaveIntEnableEx

功能	使能单个 I2C 从机的中断源
函数原型	void I2CSlaveIntEnableEx（uint32_t ui32Base，uint32_t ui32IntFlags）
参数	ui32Base　I2C 从机模块的基地址 ui32IntFlags　待清除中断源的屏蔽位
描述	该函数使能指定 I2C 从机的中断源。只有使能这个中断源才会反映到处理器中断；禁止的中断源不会影响处理器 参数 uiIntFlags 为下值任意的逻辑或： ➢ I2C_SLAVE_INT_RX ➢ _FIFO_FULL　//RX FIFO 满中断 ➢ I2C_SLAVE_INT_TX_FIFO_EMPTY　//TX FIFO 空中断 ➢ I2C_SLAVE_INT_RX_FIFO_REQ　//RX FIFO 请求中断 ➢ I2C_SLAVE_INT_TX_FIFO_REQ　//TX FIFO 请求中断 ➢ I2C_SLAVE_INT_TX_DMA_DONE　//TX DMA 完成中断 ➢ I2C_SLAVE_INT_RX_DMA_DONE　//RX DMA 完成中断 ➢ I2C_SLAVE_INT_STOP　//停止检测条件中断 ➢ I2C_SLAVE_INT_START　//开始检测条件中断 ➢ I2C_SLAVE_INT_DATA　//数据中断
返回	无

表 F-49　I2CslaveIntStatus

功能	获取当前 I2C 从机的中断状态
函数原型	bool　I2CSlaveIntStatus（uint32_t ui32Base, bool　bMasked）
参数	ui32Base　I2C 从机模块的基地址 bMasked　若要求原始的中断状态，则 bMasked 为 false；若需要屏蔽的中断状态，则 bMasked 为 true
描述	该函数返回 I2C 从机模块的中断状态。可返回原始中断状态或允许反映到处理器中的中断状态
返回	返回当前的中断状态，如果激活返回 true，如果未激活返回 false

表 F-50　I2CslaveIntStatusEx

功能	获取当前 I2C 从机的中断状态
函数原型	uint32_t I2CSlaveIntStatusEx（uint32_t ui32Base, bool bMasked）

（续）

参数	ui32Base　I2C 从机模块的基地址 bMasked　若要求原始的中断状态，则 bMasked 为 false；若需要屏蔽的中断状态，则 bMasked 为 true
描述	该函数返回 I2C 从机模块的中断状态。可返回原始中断状态或允许反映到处理器中的中断状态
返回	返回当前中断状态，枚举 I2CSlaveIntEnableEx( ) 中描述的位字段的值

表 F-51　I2CSlaveStatus

功能	获取 I2C 从机模块的状态
函数原型	uint32_t I2CSlaveStatus（uint32_t ui32Base）
参数	ui32Base　I2C 主机模块的基地址
描述	这个函数返回主机请求的动作，可能的取值如下： ➢ I2C_SLAVE_ACT_NONE ➢ I2C_SLAVE_ACT_RREQ ➢ I2C_SLAVE_ACT_TREQ ➢ I2C_SLAVE_ACT_RREQ_FBR ➢ I2C_SLAVE_ACT_OWN2SEL ➢ I2C_SLAVE_ACT_QCMD ➢ I2C_SLAVE_ACT_QCMD_DATA
返回	返回值如下： ➢ I2C_SLAVE_ACT_NONE　//没有任何 I2C 从机模块的操作被请求 ➢ I2C_SLAVE_ACT_RREQ　//I2C 主机已经把数据发送给 I2C 从机模块 ➢ I2C_SLAVE_ACT_TREQ　//I2C 主机已经请求 I2C 从机模块发送数据 ➢ I2C_SLAVE_ACT_RREQ_FBR　//I2C 主机已经发送数据到 I2C 从机；并且已接 收 //到跟在从机自身地址后的第一个字节 ➢ I2C_SLAVE_ACT_OWN2SEL　//第二个 I2C 从机地址被匹配 ➢ I2C_SLAVE_ACT_QCMD　//已收到快捷命令 ➢ I2C_SLAVE_ACT_QCMD_DATA　//在收到快捷命令时数据位已被设置

表 F-52　I2CTxFIFOConfigSet

功能	配置 I2C 发送（TX）FIFO
函数原型	void　I2CTxFIFOConfigSet（uint32_t ui32Base, uint32_t ui32Config）
参数	ui32Base　I2C 主机或从机模块的基地址 ui32Config　使用指定的宏配置 FIFO
描述	配置 I2C 外设的发送 FIFO。发送 FIFO 可被主机或从机使用，但二者不能同时使用。以下宏用于配置带或不带 DMA 的主机或从机的 TX FIFO 行为： ➢ I2C_FIFO_CFG_TX_SLAVE ➢ I2C_FIFO_CFG_TX_MASTER ➢ I2C_FIFO_CFG_TX_SLAVE ➢ I2C_FIFO_CFG_TX_MASTER_DMA ➢ I2C_FIFO_CFG_TX_SLAVE_DMA 下列的一个宏将用于选择 FIFO 的触发深度： ➢ I2C_FIFO_CFG_TX_TRIG_1 ➢ I2C_FIFO_CFG_TX_TRIG_2 ➢ I2C_FIFO_CFG_TX_TRIG_3 ➢ I2C_FIFO_CFG_TX_TRIG_4 ➢ I2C_FIFO_CFG_TX_TRIG_5 ➢ I2C_FIFO_CFG_TX_TRIG_6 ➢ I2C_FIFO_CFG_TX_TRIG_7 ➢ I2C_FIFO_CFG_TX_TRIG_8
返回	无

表 F-53　I2CTxFIFOFlush

功能	刷新发送（TX）FIFO
函数原型	void　I2CTxFIFOFlush（uint32_t ui32Base）
参数	ui32Base　I2C 主机或从机模块的基地址
描述	该函数刷新 I2C 的发送 FIFO
返回	无



附录 G　第 9 章附录：SSI 固件库函数简介

本章附录将简要介绍 SSI 的固件库函数。

表 G-1　SSIAdvDataPutFrameEnd

功能	把一个作为帧结束的数据单元存放到 SSI 发送 FIFO 中
函数原型	void　SSIAdvDataPutFrameEnd（uint32_t ui32Base，uint32_t ui32Data）
参数	ui32Base　指定 SSI 端口的基地址 ui32Data　在 SSI 接口待数据传输
描述	该函数将提供的数据保存到指定 SSI 模块的发送 FIFO 中，并标记为一个帧的结束。如果在发送 FIFO 中没有可用的空间，这个函数将一直等待直到有可用的空间方可返回。在这个字节由 SSI 模块传送完之后，应禁止 FSS 信号至少一个 SSI 时钟
返回参数	无

表 G-2　SSIAdvDataPutFrameEndNonBlocking

功能	把一个作为帧结束的数据单元存放到 SSI 发送 FIFO 中
函数原型	int32_t　SSIAdvDataPutFrameEndNonBlocking（uint32_t ui32Base， uint32_t ui32Data）
参数	ui32Base　指定 SSI 端口的基地址 ui32Data　在 SSI 接口待传输的数据
描述	该函数将提供的数据保存到指定 SSI 模块的发送 FIFO 中，并标记为一个帧的结束。在这个字节由 SSI 模块传送完之后，应禁止 FSS 信号至少一个 SSI 时钟。如果在发送 FIFO 中没有可用的空间，这个函数将返回 0
返回参数	返回写入 SSI 发送 FIFO 的单元个数

表 G-3　SSIAdvFrameHoldDisable

功能	配置 SSI 的高级模式以在每个字节传输后关闭 SSIFss 信号
函数原型	void　SSIAdvFrameHoldDisable（uint32_t ui32Base）
参数	ui32Base　SSI 端口的基地址
描述	在每个字节传输完之后，该函数使用一种高级模式配置 SSI 模块，以关闭 SSIFss 信号一个 SSI 时钟周期，这种模式为默认操作
返回参数	无

表 G-4　SSIAdvFrameHoldEnable

功能	配置 SSI 高级模式以在整个传输期间保持 SSIFss 信号
函数原型	Void　SSIAdvFrameHoldEnable（uint32_t ui32Base）
参数	ui32Base　SSI 端口的基地址

（续）

描述	该函数配置 SSI 模块使用一种高级模式以在整个数据传输中保持 SSIFss 信号。使用该模式，可直接通过 SSIAdvDataPutFrameEnd( )和 SSIAdvDataPutFrameEndNonBlocking( )来控制 SSIFss
返回参数	无

表 G-5　SSIAdvModeSet

功能	选择 SSI 模块的高级操作模式
函数原型	void　SSIAdvModeSet（uint32_t ui32Base，uint32_t　ui32Mode）
参数	ui32Base　SSI 端口的基地址 ui32Mode　使用的操作模式
描述	该函数用于选择 SSI 模块所需的高级操作模式 可供选择的模式如下： ➢ SSI_ADV_MODE_LEGACY //禁止高级模式操作，它会引起遗留或向后兼容操作。若选择该模式，将不能有效的在双或四 SPI 操作中切换。该模式为默认 ➢ SSI_ADV_MODE_WRITE　//只能将数据写入到从机的高级操作模式；任何数据会定时通过 SSIRx 引脚扔掉（而不是保存到 SSI Rx FIFO 中） ➢ SSI_ADV_MODE_READ_WRITE　//将数据写入到从机或读出从机数据的高级操作模式；该模式与 ssi_adv_mode_legacy 相同，但允许在双或四 SPI 操作中切换 ➢ SSI_ADV_MODE_BI_READ　//在双 SPI 模式中读出来自从机数据的高级操作模式，即每个 SSI 时钟读取两位数据 ➢ SSI_ADV_MODE_BI_WRITE　//在双 SPI 模式中将数据写入从机的高级操作模式，即每个 SSI 时钟写入两位数据
描述	➢ SSI_ADV_MODE_QUAD_READ　//在四 SPI 模式中读出来自从机数据的高级操作模式，即每个 SSI 时钟读取四位数据 ➢ SSI_ADV_MODE_QUAD_WRITE　//在四 SPI 模式中将数据写入从机的高级操作模式，即每个 SSI 时钟写入四位数据
返回参数	无

表 G-6　SSIBusy

功能	确认 SSI 发送器是否忙碌
函数原型	bool　SSIBusy（uint32_t ui32Base）
参数	ui32Base　SSI 端口的基地址
描述	此函数允许调用者确定是否所有的传输字节已清除了发送器硬件。如果返回 false，则发送 FIFO 为空，并且最后发送字的所有位都已从硬件移位寄存器中移出
返回参数	如果返回 true，则 SSI 正在发送；如果返回 false，则所有传输已经完成

表 G-7　SSIClockSourceGet

功能	获取指定 SSI 外设的数据时钟源
函数原型	uint32_t　SSIClockSourceGet（uint32_t ui32Base）
参数	ui32Base　SSI 端口的基地址
描述	该函数返回指定 SSI 外设的数据时钟源
返回参数	返回当前的时钟源（SSI_CLOCK_SYSTEM 或 SSI_CLOCK_PIOSC）

表 G-8　SSIClockSourceSet

功能	设置指定 SSI 外设的数据时钟源
函数原型	void　SSIClockSourceSet（uint32_t ui32Base，uint32_t ui32Source）
参数	ui32Base　SSI 端口的基地址 ui32Source　SSI 的波特率时钟源

（续）

描述	该函数允许选择 SSI 波特率的时钟源。可能的波特率时钟源为系统时钟（SSI_CLOCK_SYSTEM）或内部精确振荡器（SSI_CLOCK_PIOSC） 更改波特率时钟源会改变 SSI 产生的数据传输速率，因此更改波特率的时钟源需重新配置波特率
返回参数	无

表 G-9　SSIConfigSetExpClk

功能	配置同步串行接口（SSI ）
函数原型	void SSIConfigSetExpClk（uint32_t ui32Base, uint32_t ui32SSIClk, uint32_t ui32Protocol, uint32_t ui32Mode, uint32_t ui32BitRate, uint32_t ui32DataWidth）
参数	ui32Base　SSI 端口的基地址 ui32SSIClk　提供给 SSI 模块的时钟速率 ui32Protocol　指定数据传输协议 ui32Mode　指定操作模式 ui32BitRate　指定时钟速率 ui32DataWidt－h　指定每帧传输的位数
描述	该函数配置 SSI、设置 SSI 协议、操作模式、位速率和数据宽度 参数 ui32Protocol　定义数据帧格式，其值可选如下之一： ➢ SSI_FRF_MOTO_MODE_0 ➢ SSI_FRF_MOTO_MODE_1 ➢ SSI_FRF_MOTO_MODE_2 ➢ SSI_FRF_MOTO_MODE_3 ➢ SSI_FRF_TI ➢ SSI_FRF_NMW 摩托罗拉的帧格式编码根据以下的极性和相位配置： 极性（Polarity）　相位（Phase）　模式 0　0　SSI_FRF_MOTO_MODE_0 0　1　SSI_FRF_MOTO_MODE_1 1　0　SSI_FRF_MOTO_MODE_2 1　1　SSI_FRF_MOTO_MODE_3 参数 ui32Mode 定义了 SSI 模块的操作模式 SSI 模块既可配置为主机模式也可配置为从机模式；当作为从设备时，SSI 可配置为禁止其串行输出线输出 参数 ui32Mode 可选以下值之一： ➢ SSI_MODE_MASTER，SSI_MODE_SLAVE ➢ SSI_MODE_SLAVE_OD 参数 ui32BitRate 定义了 SSI 的比特率 该位速率必须满足以下时钟比条件： ➢ FSSI≥2＋位速率（主模式）//这个速度不能超过 25 MHz ➢ FSSI≥12＋位速率或者 6×位速率（从模式）//取决于特定微控制器的特性 其中 FSSI 是 SSI 模块提供的时钟的频率 参数 ui32DataWidth 定义了传输数据的宽度，其值为 4～16 这里，外设时钟和处理器时钟相同。如果它是一个已知常数（调用 SysCtlClockGet( )来节约编码/执行的开销），该值由 SysCtlClockGet( )返回，或者它为显式地硬编码
返回参数	无

表 G-10　SSIDataGet

功能	从 SSI 接收 FIFO 中获取数据单元
函数原型	void　SSIDataGet（uint32_t ui32Base，uint32_t　* pui32Data）

（续）

参数	ui32Base　指定 SSI 端口的基地址 pui32Data　指向 SSI 端口接收数据存储单元的指针
描述	该函数从指定 SSI 模块的接收 FIFO 中获取接收到的数据，并把数据存放到由 pui32Data 参数指定的地址中。若没有可用的数据，该函数将一直等待到接收到数据方可返回
返回参数	无

表 G-11　SSIDataGetNonBlocking

功能	从 SSI 模块接收 FIFO 中读取数据元素
函数原型	int32_t　SSIDataGetNonBlocking（uint32_t ui32Base， uint32_t * pui32Data）
参数	ui32Base　指定 SSI 端口的基地址 pui32Data　指向 SSI 端口接收数据存储单元的指针
描述	该函数从指定 SSI 模块的接收 FIFO 中获取接收到的数据，并把数据存放到由 pui32Data 参数指定的地址中。如果 FIFO 中没有数据，则该函数将返回 0
返回参数	返回从 SSI 接收 FIFO 中读取的数据单元的个数

表 G-12　SSIDataPut

功能	把数据单元存放到 SSI 发送 FIFO 中
函数原型	void　SSIDataPut（uint32_t ui32Base，uint32_t ui32Data）
参数	ui32Base　指定 SSI 端口的基地址 ui32Data　待从 SSI 端口发送的数据
描述	该函数把提供的数据存放到指定 SSI 模块的发送 FIFO 中。如果在发送 FIFO 中没有可用的数据，该函数将一直等待到接收到数据方可返回
返回参数	无

表 G-13　SSIDataPutNonBlocking

功能	把一个数据单元存放到 SSI 发送 FIFO 中
函数原型	int32_t　SSIDataPutNonBlocking（uint32_t　ui32Base， uint32_t ui32Data）
参数	ui32Base　指定 SSI 端口的基地址 ui32Data　待从 SSI 端口发送的数据
描述	该函数把提供的数据存放到指定 SSI 模块的发送 FIFO 中。如果在发送 FIFO 中没有可用的数据，则该函数将返回 0
返回参数	返回写入到 SSI 发送 FIFO 中的数据单元的个数

表 G-14　SSIDisable

功能	禁止同步串行接口
函数原型	void　SSIDisable（uint32_t ui32Base）
参数	ui32Base　SSI 端口的基地址
描述	该函数禁止同步串行接口的操作
返回参数	无

表 G-15　SSIDMADisable

功能	禁止 SSI DMA 操作
函数原型	void　SSIDMADisable（uint32_t ui32Base uint32_t ui32DMAFlags）
参数	ui32Base　SSI 端口的基地址 ui32DMAFlags　禁止 DMA 功能的位屏蔽
描述	该函数用于禁止由 SSIDMAEnable( )使能的指定 SSI DMA 功能 参数 ui32DMAFlags 是下列任意值的逻辑或： ➢ SSI_DMA_RX　// 禁止 DMA 接收 ➢ SSI_DMA_TX　// 禁止 DMA 发送
返回参数	无

表 G-16　SSIDMAEnable

功能	使能 SSI DMA 操作
函数原型	void　SSIDMAEnable（uint32_t ui32Base，uint32_t ui32DMAFlags）
参数	ui32Base　SSI 端口的基地址 ui32DMAFlags　使能 DMA 功能的位屏蔽
描述	该函数使能指定 SSI DMA 功能。SSI 可以配置为使用 DMA 进行数据发送和/或接收 参数 ui32DMAFlags 为下列任意值的逻辑或： ➢ SSI_DMA_RX　// 使能 DMA 接收 ➢ SSI_DMA_TX　// 使能 DMA 发送
返回参数	无

表 G-17　SSIEnable

功能	使能同步串行接口模块
函数原型	void SSIEnable（uint32_t ui32Base）
参数	ui32Base　指定 SSI 端口的基地址
描述	该函数使能 SSI 模块的操作，SSI 必须在使能前配置
返回参数	无

表 G-18　SSIIntClear

功能	清除 SSI 中断来源
函数原型	void　SSIIntClear（uint32_t ui32Base，uint32_t ui32IntFlags）
参数	ui32Base　指定 SSI 端口的基地址 ui32IntFlags　待清除中断源的位屏蔽
描述	清除指定 SSI 的中断源，使之不再有效。该函数必须在中断处理程序调用，以防止在退出时中断再次被触发 参数 ui32IntFlags 可包含 SSI_RXTO 与 SSI_RXOR 值的一个或两个
返回参数	无

表 G-19　SSIIntDisable

功能	禁止单个的 SSI 中断源
函数原型	void SSIIntDisable（uint32_t ui32Base，uint32_t ui32IntFlags）
参数	ui32Base　指定 SSI 端口的基地址 ui32IntFlags　禁止中断源的位屏蔽

（续）

描述	该函数禁止指示的 SSI 中断源 参数 ui32IntFlags 可以是以下任一值： ➢ SSI_TXFF ➢ SSI_RXFF ➢ SSI_RXTO ➢ SSI_RXOR
返回参数	无

表 G-20　SSIIntEnable

功能	使能单个 SSI 模块的中断源
函数原型	void　SSIIntEnable（uint32_t ui32Base，uint32_t ui32IntFlags）
参数	ui32Base　指定 SSI 端口的基地址 ui32IntFlags　使能中断源的位屏蔽
描述	该函数使能 SSI 中断源指示。只有被使能的中断源才能反应到处理器中断中；被禁止的中断源对处理器没有影响 参数 ui32IntFlags 为以下任一值： ➢ SSI_TXFF ➢ SSI_RXFF ➢ SSI_RXTO ➢ SSI_RXOR
返回参数	无

表 G-21　SSIIntRegister

功能	给 SSI 注册一个中断处程序
函数原型	void SSIIntRegister（uint32_t ui32Base，void（ * pfnHandler）（void））
参数	ui32Base　指定 SSI 端口的基地址 pfnHandler　指向 SSI 中断发生时被调函数的指针
描述	当产生 SSI 中断时，该函数将注册一个被调用的中断处理程序。该函数使能中断控制器中的全局中断；特定 SSI 中断必须由 SSIIntEnable( ) 函数使能。若需要，中断处理程序还将使用 SSIIntClear( ) 来清除中断源
返回参数	无

表 G-22　SSIIntStatus

功能	获取当前的中断状态
函数原型	uint32_t SSIIntStatus（uint32_t ui32Base， bool bMasked）
参数	ui32Base　指定 SSI 端口的基地址 bMasked　如果要求的为原始中断状态，则 bMasked 为 False；若所需的为屏蔽中断状态，则 bMasked 为 True
描述	该函数返回指定 SSI 模块的中断状态。返回的是原始中断状态或允许反映到处理器的中断状态
返回参数	返回当前中断状态，并通过以下的位字段枚举出来： ➢ SSI_TXFF ➢ SSI_RXFF ➢ SSI_RXTO SSI_RXOR

表 G-23　SSIIntUnregister

功能	注销 SSI 的中断处理程序
函数原型	void SSIIntUnregister（uint32_t ui32Base）
参数	ui32Base　指定 SSI 端口的基地址
描述	在产生 SSI 中断时，该函数将清除被调用的中断处理程序。这个函数同时也禁止了中断控制器中的中断，使中断处理程序不会再被调用
返回参数	无



附录 H　第 10 章附录：内部存储器的固件库函数简介

本章附录将要介绍的内部存储器固件库函数包括：

- H. 1 闪存（Flash）的固件库函数
- H. 2 内存保护单元（MPU）的固件库函数
- H. 3 EEPROM 固件库函数

H. 1　闪存（Flash）固件库函数

表 H-1　FlashErase

功能	擦除一个闪存的区块
函数原型	int32_t FlashErase（uint32_t ui32Address）
参数	ui32Address　待擦除的闪存块的起始地址
描述	该函数将删除 1KB 的片上闪存区块。闪存在擦除该区块之后必须被填充 0xFF 字节，但只读与只执行区块不能被擦除。且只有当区块被擦除后，该函数才能返回
返回参数	返回 0 表示成功，-1 表示指定的块地址无效，或者块被写保护

表 H-2　FlashIntClear

功能	清除闪存控制器的中断源
函数原型	void FlashIntClear（uint32_t　ui32IntFlags）
参数	ui32IntFlags　待清除中断源的位屏蔽，其值可取 FLASH_INT_PROGRAM 或 FLASH_INT_AMISC
描述	清除指定闪存控制器的中断源，使之不再有效。该函数必须在中断处理程序中被调用，对其进行调用。以防止在退出时又立即被再次被触发
返回参数	无

表 H-3　FlashIntDisable

功能	禁止单个闪存控制器的中断源
函数原型	void FlashIntDisable（uint32_t ui32IntFlags）
参数	ui32IntFlags　待禁止中断源的位屏蔽。其值可取 FLASH_INT_PROGRAM 或 FLASH_INT_ACCESS
描述	该函数禁止指定闪存控制器的中断源。只有使能的中断源才能被反映到处理器中断；禁止的中断源对处理器无影响
返回参数	无

表 H-4　FlashIntEnable

功能	使能单个闪存控制器的中断源
函数原型	voidFlashIntEnable（uint32_t ui32IntFlags）
参数	ui32IntFlags　待使能中断源的位屏蔽。其值可取 FLASH_INT_PROGRAM 或 FLASH_INT_ACCESS
描述	该函数使能指定闪存控制器的中断源。只有使能的中断源才能被反映到处理器中断；禁止的中断源对处理器无影响
返回参数	无

表 H-5　FlashIntRegister

功能	注册闪存中断的中断处理程序
函数原型	void FlashIntRegister（void（ * pfnHandler）（void））
参数	pfnHandler　指向在闪存中断发生时被调函数的指针
描述	在闪存中断发生时，该函数用于设置被调用的中断处理程序。当无效闪存访问发生时，闪存控制器将会发出一个中断。在编程或擦除操作完成时，也可产生中断。并且在注册中断处理程序时会自动使能该中断
返回参数	无

表 H-6　FlashIntStatus

功能	获取当前的中断状态
函数原型	uint32_t FlashIntStatus（bool bMasked）
参数	bMasked　若要求原始中断状态，则 bMasked 为 false；如果所需的是屏蔽中断状态，则 bMasked 为 true
描述	该函数将返回闪存控制器的中断状态。返回原始中断状态或允许反映到处理器中的中断状态
返回参数	返回当前的中断状态，而以如下的位字段枚举： ➢ FLASH_INT_PROGRAM ➢ FLASH_INT_ACCESS

表 H-7　FlashIntUnregister

功能	注销闪存中断的中断处理程序
函数原型	void FlashIntUnregister（void）
描述	该函数清除闪存中断发生时被调用的中断处理程序。该函数也屏蔽了中断控制器中的中断，以使中断处理程序不再被调用
返回参数	无

表 H-8　FlashProgram

功能	闪存编程
函数原型	int32_t FlashProgram（uint32_t　* pui32Data， uint32_t ui32Address， uint32_t ui32Count）
参数	pui32Data　指向待编程数据的指针 ui32Address　待编程闪存的开始地址，其必须为 4 的倍数 ui32Count　待编程的字节数，其必须为 4 的倍数。

（续）

描述	该函数将一组字编程到片上闪存中。闪存页中的每个字在每次擦除该页后只可以编程一次；多次编程一个字将导致该字所在的闪存中出现不可预期的值。因为闪存一次只能编程一个字，且起始地址和字节数都必须为 4 的倍数。若需要，将由调用者来验证编程的内容 该函数只有在数据编程结束之后方可返回
返回参数	成功返回 0；若遇到编程错误，则返回 -1

表 H-9　FlashProtectGet

功能	获取闪存块保护的设置
函数原型	tFlashProtection　FlashProtectGet（uint32_t　ui32Address）
参数	ui32Address　待查询闪存块的起始地址
描述	该函数获取指定 2KB 闪存块的当前保护。每个块可被读/写、只读或只执行。读/写块可被读取、执行、擦除和编程；只读块可被读取与执行；只执行块只可被执行，而不允许处理器和调试器读取数据
返回参数	返回此块的保护设置，可能的取值请参考 FlashProtectSet( )

表 H-10　FlashProtectSave

功能	保存闪存的保护设置
函数原型	int32_t FlashProtectSave（void）
描述	该函数将保存当前编程闪存保护的永久配置。在一些器件上，该操作是不可逆的；而芯片复位或电源周期不会改变闪存的保护 只有在保存保护设置以后，该函数方可返回
返回参数	0 表示成功，-1 表示遇到硬件错误

表 H-11　FlashProtectSet

功能	设置闪存区块的保护设置
函数原型	int32_t FlashProtectSet（uint32_t ui32Address，tFlashProtection eProtect）
参数	ui32Address　待保护的闪存区块的起始地址 eProtect　应用于块的保护。取值为下列之一： ➢ FlashReadWrite ➢ FlashReadOnly ➢ FlashExecuteOnly
描述	该函数为指定的 2KB 闪存块设置保护。读/写块可被设置成只读或只执行；只读块也可设置成只执行；但只执行块不能修改已有的保护设置
返回参数	0 表示成功，-1 表示指定的地址或保护设置无效

表 H-12　FlashUserGet

功能	获取用户寄存器
函数原型	int32_t FlashUserGet（uint32_t　*pui32User0， uint32_t　*pui32User1）
参数	pui32User0　指向存放用户寄存器 0 单元的指针 pui32User1　指向存放用户寄存器 1 单元的指针
描述	该函数读取用户寄存器（0 和 1）的内容，并将其保存到指定的单元
返回参数	返回 0 表示成功，-1 表示遇到硬件错误

表 H-13　FlashUserSave

功能	保存用户寄存器
函数原型	int32_t FlashUserSave（void）
描述	该函数将保存当前编程闪存保护的永久配置。在一些器件上，该操作是不可逆的；芯片复位或电源周期不会改变闪存的保护
返回参数	无

表 H-14　FlashUserSet

功能	设定用户寄存器
函数原型	int32_t FlashUserSet（uint32_t ui32User0，uint32_t ui32User1）
参数	ui32User0　存储在用户寄存器 0 的值 ui32User1　存储在用户寄存器 1 的值
描述	该函数将用户寄存器的（0 和 1）内容设置为指定的值
返回参数	返回 0 表示成功，-1 表示遇到硬件错误

H. 2　内存保护单元（MPU）固件库函数

本小节将介绍 MPU 的固件库函数的功能。

表 H-15　MPUDisable

功能	禁止使用 MPU
函数原型	voidMPUDisable（void）
描述	该函数禁用 Cortex-M 内存保护单元。当禁止 MPU 时，使用默认的存储器映射且不产生存储器管理故障
返回参数	无

表 H-16　MPUEnable

功能	使能和配置 MPU 的使用
函数原型	voidMPUEnable（uint32_t ui32MPUConfig）
参数	ui32MPUConfig　可能配置的逻辑或
描述	该函数使能 Cortex-M 内存保护单元。配置特权模式下的默认行为，同时处理硬故障或NMI。在使能 MPU 之前，至少必须调用 MPURegionSet()来设置一个区域，不然，将 MPU_CONFIG_PRIV_DEFAULT 标志传递给 MPUEnable()来使能特权模式下的默认区域。一旦使能 MPU，一切违规内存访问，都将产生存储器管理故障 参数 ui32MPUConfig 应为以下任意值的逻辑或： ➢ MPU_CONFIG_PRIV_DEFAULT//在特权模式下与无其他区域定义时使能默认存储器映射。当使能 MPU 时，如果未开启此选项，则必须定义至少一个有效区域 ➢ MPU_CONFIG_HARDFLT_NMI//在发生硬故障或 NMI 异常处理时使能 MPU。如果未开启此选项，当其中的一个异常处理程序和应用了默认的内存映射时，将禁止 MPU ➢ MPU_CONFIG_NONE//不选择上述选项。在这种情况下，特权模式不提供默认的存储器映射，并且在故障处理程序中 MPU 不会被使能
返回参数	无

表 H-17　MPUIntRegister

功能	注册存储器管理故障的中断处理程序
函数原型	voidMPUIntRegister（void（*pfnHandler）（void））

（续）

参数	pfnHandler　指向在存储器管理故障发生时被调函数的指针
描述	非法访问保护区将使 MPU 产生存储器管理故障，该函数用于设置和使能被调用的处理程序
返回参数	无

表 H-18　MPUIntUnregister

功能	注销存储器管理故障的中断处理程序
函数原型	void MPUIntUnregister（void）
描述	该函数将禁止和清除在存储器管理故障发生时被调用的处理程序
返回参数	无

表 H-19　MPURegionCountGet

功能	获取 MPU 支持的区域数目
函数原型	uint32_tMPURegionCountGet（void）
描述	该函数用于获取 MPU 所支持的区域总数，包括已编程的区域
返回参数	使用 MPURegionSet（） 进行编程的可用存储器保护区的数量

表 H-20　MPURegionDisable

功能	禁止指定的区域
函数原型	voidMPURegionDisable（uint32_t ui32Region）
参数	ui32Region　禁止的区域号
描述	该函数用于禁止先前使能的内存保护区域。如果未另外调用 MPURegionSet（）来覆盖该区域，则此区域的配置仍将保留，并且可通过调用 MPURegionEnable（）再次使能该区域
返回参数	无

表 H-21　MPURegionEnable

功能	使能指定区域
函数原型	voidMPURegionEnable（uint32_t ui32Region）
参数	ui32Region　使能的区域号
描述	该函数用于使能存储器保护区域。该区域已使用 MPURegionSet（）函数配置过。一旦使能，该区域的存储器保护规则将被开启，非法访问将导致存储器管理故障
返回参数	无

表 H-22　MPURegionGet

功能	获取当前指定区域的设置
函数原型	voidMPURegionGet（uint32_t ui32Region, uint32_t ＊pui32Addr, uint32_t ＊pui32Flags）
参数	ui32Region　获取的区域号 pui32Addr　指向区域基地址存放单元的指针 pui32Flags　指向区域属性标志的指针
描述	该函数获取指定区域的配置。参数的含义和格式与 MPURegionSet（）函数相同 此函数用于保存供以后使用的 MPURegionSet（）函数的区域配置。该区域的使能状态被保存在已被保存的属性中
返回参数	无

表 H-23　MPURegionSet

功能	设置某一特定区域的访问规则
函数原型	void MPURegionSet（uint32_t ui32Region, uint32_t ui32Addr, uint32_t ui32Flags）
参数	ui32Region　待设置的区域号 ui32Addr　区域的基地址。它必须按照 ui32Flags 指定区域的大小对齐 ui32Flags　一组定义区域的属性的标志
描述	该函数用于设置一个区域的保护规则。该区域具有一个基址、一组属性，以及区域的大小（它必须为 2 的幂），其中，基址参数 ui32Addr 必须根据区域大小对齐 参数 ui32Flags 是所有区域属性的逻辑或。它是区域大小、执行权限、读/写权限、禁止子区域以及标志的结合选择来决定是否使能该区域 标志大小决定区域的大小，并且必须为以下之一： ➢ MPU_RGN_SIZE_32B ➢ MPU_RGN_SIZE_64B ➢ MPU_RGN_SIZE_128B ➢ MPU_RGN_SIZE_256B ➢ MPU_RGN_SIZE_512B ➢ MPU_RGN_SIZE_1K ➢ MPU_RGN_SIZE_2K ➢ MPU_RGN_SIZE_4K ➢ MPU_RGN_SIZE_8K ➢ MPU_RGN_SIZE_16K ➢ MPU_RGN_SIZE_32K ➢ MPU_RGN_SIZE_64K ➢ MPU_RGN_SIZE_128K ➢ MPU_RGN_SIZE_256K ➢ MPU_RGN_SIZE_512K ➢ MPU_RGN_SIZE_1M ➢ MPU_RGN_SIZE_2M ➢ MPU_RGN_SIZE_4M ➢ MPU_RGN_SIZE_8M ➢ MPU_RGN_SIZE_16M ➢ MPU_RGN_SIZE_32M ➢ MPU_RGN_SIZE_64M ➢ MPU_RGN_SIZE_128M ➢ MPU_RGN_SIZE_256M ➢ MPU_RGN_SIZE_512M ➢ MPU_RGN_SIZE_1G ➢ MPU_RGN_SIZE_2G ➢ MPU_RGN_SIZE_4G 执行权限标志必须是下列之一： ➢ MPU_RGN_PERM_EXEC//使能代码执行的区域 ➢ MPU_RGN_PERM_NOEXEC　//禁止代码执行的区域 在特权模式与用户模式下，可单独应用读/写访问权限。读/写访问标志必须为下列之一： ➢ MPU_RGN_PERM_PRV_NO_USR_NO //无特权或用户模式访问权限 ➢ MPU_RGN_PERM_PRV_RW_USR_NO//特权读/写，用户不能访问 ➢ MPU_RGN_PERM_PRV_RW_USR_RO//特权读/写，用户只读 ➢ MPU_RGN_PERM_PRV_RW_USR_RW//特权读/写，用户可读/写 ➢ MPU_RGN_PERM_PRV_RO_USR_NO//特权只读，用户不能访问 ➢ MPU_RGN_PERM_PRV_RO_USR_RO//特权只读，用户只读 由 MPU 将区域自动分割成 8 个大小相等的子区域。子区域仅用于≥256B 的区域。这 8 个子区域的任何一个都可被禁止，允许创建的“洞”在一个区域可被悬空，或被具有不同属性的另一区域所覆盖。8 个子区域中的任何一个可被下列任何标志的逻辑或所禁止： ➢ MPU_SUB_RGN_DISABLE_0 ➢ MPU_SUB_RGN_DISABLE_1

（续）

描述	➢ MPU_SUB_RGN_DISABLE_2 ➢ MPU_SUB_RGN_DISABLE_3 ➢ MPU_SUB_RGN_DISABLE_4 ➢ MPU_SUB_RGN_DISABLE_5 ➢ MPU_SUB_RGN_DISABLE_6 ➢ MPU_SUB_RGN_DISABLE_7 可用下列之一的标志开始使能或禁止区域： ➢ MPU_RGN_ENABLE ➢ MPU_RGN_DISABLE
返回参数	无

H.3 EEPROM 固件库函数

本小节将以定义文档和函数文档两部分来介绍 EEPROM 的固件库函数。

1. 定义文档

表 H-24　EEPROM_INIT_ERROR

定义	#define　EEPROM_INIT_ERROR
描述	该值可从调用 EEPROMInit ( ) 函数返回。它这表明先前的数据或写保护操作将被复位事件所中断，并且在这问题发生之后将使 EEPROM 外设无法被清除。但该情况可用另一次复位来解决或者严重依赖问题的起因。例如，如果部分电压不稳定，一旦电压稳定之后便可清除该错误

表 H-25　EEPROM_INIT_OK

定义	#define　EEPROM_INIT_OK
描述	该值可从调用 EEPROMInit ( ) 函数返回。它表明先前的写操作没有被复位事件所中断，并且该 EEPROM 外设已可被使用

表 H-26　EEPROM_ INIT_RETRY

定义	#define　EEPROM_INIT_RETRY
描述	此值可从调用 EEPROMInit ( ) 函数返回。它这表明先前的数据或写保护操作将被复位事件所中断。虽然 EEPROM 外设已经恢复其状态，但上次的写操作可能被丢失。因此应用程序必须检查已写入数据的有效性，并重试任何需要的写操作

表 H-27　EEPROM_ INT_PROGRAM

定义	#define　EEPROM_INT_PROGRAM
描述	如果正在发出一个 EEPROM 中断的信号，这个值可以传递给 EEPROMIntEnable( ) 和 EEPROMIntDisable( )，并由 EEPROMIntStatus ( ) 返回

表 H-28　EEPROM_PROT_NA_LNA_URW

定义	#define　EEPROM_PROT_NA_LNA_URW
描述	该值可传递给 EEPROMBlockProtectSet( ) 或从 EEPROMBlockProtectGet( ) 返回。它表示该块既不提供读也不提供写访问，除非它是受密码保护和解锁

表 H-29　EEPROM_PROT_RO_LNA_URO

定义	#define　EEPROM_PROT_RO_LNA_URO
描述	该值可传递给 EEPROMBlockProtectSet( ) 或从 EEPROMBlockProtectGet( ) 返回。它表示该块应在未设置密码时或已设置密码但块被解锁时提供只读访问。当已设置密码且块被锁定，则读/写操作将被禁止

表 H-30　EEPROM_PROT_RW_LRO_URW

定义	#define　EEPROM_PROT_RW_LRO_URW
描述	该值可传递给 EEPROMBlockProtectSet( ) 或从 EEPROMBlockProtectGet( ) 返回。它表示当未设置密码或设置了密码但该块已被解锁时，该块应提供读/写访问，以及设置了密码但块锁定时可提供只读访问

表 H-31　EEPROM_PROT_SUPERVISOR_ONLY

定义	#define　EEPROM_PROT_SUPERVISOR_ONLY
描述	这一位可与保护选项作或运算后传递给 eepromblockprotectset( )，或从 eepromblockprotect-get( ) 返回。在管理员模式下限制运行 EEPROM 访问的线程，以及当 CPU 处于用户模式时，阻止访问 EEPROM 模块

表 H-32　EEPROM_RC_INVPL

定义	#define　EEPROM_RC_INVPL
描述	该返回码位（code bit）表示 EEPROM 编程状态机写入失败（由于低于 EEPROM 编程所需的电压）。一旦电压稳定该操作可能会重试

表 H-33　EEPROM_RC_NOPERM

定义	#define　EEPROM_RC_NOPERM
描述	该返回码位表示试图写入一个值，但是目标权限不允许写操作。这可能是由于目标块被锁定、访问保护设置为禁止写入，或尝试写一个已经存在的密码

表 H-34　EEPROM_RC_WKCOPY

定义	#define　EEPROM_RC_WKCOPY
描述	该返回码位表示 EEPROM 编程状态机当前正复制到或从内部复制缓冲区，为新值的写入创建空间。这可作为一个状态指示器，但不显示错误

表 H-35　EEPROM_RC_WKERASE

定义	#define　EEPROM_RC_WKERASE
描述	该返回码位表示 EEPROM 编程状态机当前正在擦除内部复制的缓冲区。这可作为一个状态指示器，但不显示错误

表 H-36　EEPROM_RC_WORKING

定义	#define　EEPROM_RC_WORKING
描述	该返回码位表示 EEPROM 编程状态机正在工作。直到清除该位才能尝试新的写操作

表 H-37　EEPROM_RC_WRBUSY

定义	#define　EEPROM_RC_WRBUSY
描述	该返回码位表示在写操作进行时，试图从 EEPROM 中做读操作

表 H-38　EEPROMAddrFromBlock

功能	返回 EEPROM 模块中第一个字的偏移地址
定义	#defineEEPROMAddrFromBlock（ui32Block）

（续）

参数	ui32Addr　返回 EEPROM 模块中第一个字地址的索引
描述	该宏可被用于确定给定 EEPROM 模块中第一个字的地址。返回的地址表示 EEPROM 存储基址的字节偏移量
返回	返回给定 EEPROM 模块第一个字的地址

表 H-39　EEPROMBlockFromAddr

功能	返回包含给定偏移地址的 EEPROM 块号
定义	#defineEEPROMBlockFromAddr（ui32Addr）
参数	ui32Addr　返回线性的、EEPROM 存储单元的字节地址块号。这是一个从 EEPROM 起始存储单元的零偏移量
描述	该宏可用于把 EEPROM 地址偏移转换成一个块号，以适用于任何驱动程序的块保护函数中。提供的地址表示 EEPROM 基地址的字节偏移量
返回	返回包含传递地址从零开始的块号

2. 函数文档

表 H-40　EEPROMBlockCountGet

功能	确定在 EEPROM 中块的数量
函数原型	uint32_tEEPROMBlockCountGet（void）
描述	该函数可用来确定在 EEPROM 中块的数量。每个块有相同的大小，在一个块中包含的存储字节数由设备分割的块大小决定，这可通过调用 EEPROMSizeGet ( ) 函数来获取，并通过该函数返回块的数量
返回	返回 EEPROM 中块的总数

表 H-41　EEPROMBlockHide

功能	在下次复位前，隐藏 EEPROM 块
函数原型	void EEPROMBlockHide（uint32_t ui32Block）
参数	ui32Block　待隐藏的 EEPROM 块号
描述	该函数隐藏块 0 以外的 EEPROM 块。一旦隐藏，块在下次复位前是完全无法访问的。这种机制允许初始化代码访问被隐藏的应用程序其余部分的数据。不同于使用密码的应用程序，它使用的块隐藏不需要包含任何可通过反汇编发现的嵌入密码
返回	无

表 H-42　EEPROMBlockLock

功能	锁定受密码保护的 EEPROM 块
函数原型	uint32_tEEPROMBlockLock（uint32_t ui32Block）
参数	ui32Block　待锁定的 EEPROM 块号
描述	该函数用于锁定已被先前写入密码保护的 EEPROM 块。访问被锁定的块由以前调用的 EEPROMBlockProtectSet ( ) 函数应用的保护设置来决定。如果这个块先前没有设置密码，该函数没有任何影响。 锁定块 0 具有使 EEPROM 中的所有其他块不可访问的效果
返回	返回退出块锁定状态。如果未锁定返回 1；如果锁定则返回 0

表 H-43　EEPROMBlockPasswordSet

功能	设置用于保护 EEPROM 块的密码
函数原型	uint32_tEEPROMBlockPasswordSet（uint32_t ui32Block, uint32_t * pui32Password, uint32_t ui32Count）
参数	ui32Block　待设置密码的 EEPROM 块号 pui32Password　指向由 uint32_t 值构成的密码设置数组的指针。每个元素可以是除了 0xFFFFFFFF 以外的任何 32 位值。这个数组必须包含由 ui32Count 参数给定的元素数量 ui32Count　提供 ui32Password 中 uint32_t 的数量，有效的值为 1、2、3
描述	该函数允许密码用于解锁被设置的 EEPROM 块。有效密码由 32、64 或 96 位任意值的字组成（0xFFFFFFFF 除外）。密码只能设置一次，任何进一步尝试设置密码的操作将导致错误。一旦设置了密码，该块或块 0 在调用 EEPROMBlockLock( )或发生复位以前，该块将一直保持解锁状态 如果密码设置在块 0 上，这会影响整个外设的锁定。当锁定块 0 时，在块 0 解锁前，所有其他 EEPROM 块将无法访问。一旦块 0 解锁，可根据设置在其他块上的密码来访问相应的块，该块的保护设置可通过调用 EEPROMBlockProtectSet( )来实现
返回	返回下列一个逻辑或组合来表示状态和错误条件： ➢ EEPROM_RC_INVP ➢ EEPROM_RC_WRBUSY ➢ EEPROM_RC_NOPERM ➢ EEPROM_RC_WKCOPY ➢ EEPROM_RC_WKERASE ➢ EEPROM_RC_WORKING

表 H-44　EEPROMBlockProtectGet

功能	返回 EEPROM 块的当前保护等级
函数原型	uint32_tEEPROMBlockProtectGet（uint32_t ui32Block）
参数	ui32Block　待查询的保护等级的块号
描述	该函数返回当前给定 EEPROM 块的保护设置。若当前锁定块 0，则必须先解锁块 0，然后才能调用这个函数来查询其他块的保护设置
返回	返回括号内任选其中一个值（EPROM_PROT_RW_LRO_URW、EEPROM_PROT_NA_LNA_URW、EEPROM_PROT_RO_LNA_URO）与 EEPROM_PROT_SUPERVISOR_ONLY 值的或运算

表 H-45　EEPROMBlockProtectSet

功能	设置 EEPROM 块的当前保护选项
函数原型	uint32_tEEPROMBlockProtectSet（uint32_t ui32Block，uint32_t ui32Protect）
参数	ui32Block　待设置保护选项的块号 ui32Protect　任选括号内其中一个值（EEPROM_PROT_RW_LRO_URW、EEPROM_PROT_NA_LNA_URW、EEPROM_PROT_RO_LNA_URO）与 EEPROM_PROT_SUPERVISOR_ONLY 值的或运算

（续）

描述	该函数设置给定 EEPROM 块的保护设置（假设先前没有保护设置被写入）。注意：块 1 及以上的保护设置是叠加在块 0 的保护设置之上的，即有效的保护设置是块 0 保护标志位与目标块保护标志位的逻辑或。此协议允许通过块 0 设置整个设备的全局保护选项，更严格的保护设置可逐块设置 保护标志指示的访问权限如下： ➢ EEPROM_PROT_SUPERVISOR_ONLY //在管理员模式下限制访问块的线程运行。若清除此标志，用户和管理员线程都可以访问该块 ➢ EEPROM_PROT_RW_LRO_URW //若块未设置密码，或即使设置了密码但块已被解锁，可提供块的读/写访问。若该块已被锁定，则仅允许只读访问 ➢ EEPROM_PROT_NA_LNA_URW //既不提供读也不提供写访问，除非设置了密码的块被解锁。若块已解锁，则允许该块的读取和写入访问 ➢ EEPROM_PROT_RO_LNA_URO //若块未设置密码，或块设置了密码但已被解锁，则提供块的读访问。如果块设置了密码保护且已被锁定，则不允许访问该块
返回	返回下列值组合的一个逻辑或来指示状态和错误条件： ➢ EEPROM_RC_INVPL ➢ EEPROM_RC_WRBUSY ➢ EEPROM_RC_NOPERM ➢ EEPROM_RC_WKCOPY ➢ EEPROM_RC_WKERASE ➢ EEPROM_RC_WORKING

表 H-46　EEPROMBlockUnlock

功能	解锁密码保护的 EEPROM 块
函数原型	uint32_tEEPROMBlockUnlock（uint32_t ui32Block, uint32_t ＊pui32Password, uint32_t ui32Count）
参数	ui32Block　待解锁的 EEPROM 块号 pui32Password　指向由 uint32_t 值构成的密码设置数组的指针。可通过调用 EEPROM-BlockPasswordSet（）来使每个元素都必须与原始设置的密码匹配 ui32Count　提供 pui32Password 数组中的元素数目，且必须与原始传递给 EEPROMBlock-PasswordSet（）中的值匹配。有效的值为 1、2、3
描述	该函数用于解锁 EEPROM 块先前写入的密码保护。一旦块被解锁，块的访问就取决于之前调用 EEPROMBlockProtectSet（）函数所设置的保护 要成功解锁 EEPROM 块，提供的密码必须与 EEPROMBlockPasswordSet（）中设置的原始密码匹配。若密码不正确，块将保持锁定 解锁块 0 将使所有其他块可根据自己的访问保护设置访问设备。当块 0 锁定时，所有其他 EEPROM 块将不能被访问
返回	返回退出时的锁定状态，如果返回 1 代表解锁，如果返回 0 表示锁定

表 H-47　EEPROMInit

功能	在写操作时遭遇电源故障所执行的任何必要的恢复
函数原型	uint32_tEEPROMInit（void）
描述	该函数必须在 SysCtlPeripheralEnable（）之后，以及在访问 EEPROM 前调用，以检查以前在写操作时由于电源故障造成的错误。该函数检测这些错误，并在返回信息给调用者之前尽可能的恢复这些错误，无论先前的数据是否丢失都必须重试 返回 EEPROM_INIT_RETRY，应用程序负责确定哪个写数据可能已丢失，并重写这个数据。如果返回 EEPROM_INIT_ERROR，则 EEPROM 将无法恢复其状态。这个状况能否通过未来的复位解决取决于故障原因。例如，若因电源电压不稳定，在电压稳定后，重试操作可能会清除该错误。在复位后错误的调用该函数可能导致错误的操作，若 EEPROM 中的数据是以后写入的可能使其永久丢失

（续）

返回	若未检测到错误，则返回 EEPROM_INIT_OK；如果先前写操作被电源或复位事件所中断，则返回 EEPROM_INIT_RETRY；若 EEPROM 外设目前无法恢复中断的写入或擦除操作，则返回 EEPROM_INIT_ERROR

表 H-48　EEPROMIntClear

功能	清除 EEPROM 中断
函数原型	voidEEPROMIntClear（uint32_t ui32IntFlags）
参数	ui32IntFlags　指示待清除的中断源，当前唯一的有效值为 EEPROM_INT_PROGRAM
描述	该函数允许应用程序清除 EEPROM 中断
返回	无

表 H-49　EEPROMIntDisable

功能	禁止 EEPROM 中断
函数原型	voidEEPROMIntDisable（uint32_t ui32IntFlags）
参数	ui32IntFlags　指示待禁止的 EEPROM 中断源，当前必须为 EEPROM_INT_PROGRAM
描述	该函数禁止 EEPROM 中断，以及防止当任何 EEPROM 写或擦除操作完成时调用中断向量。EEPROM 外设共享一个单一的中断向量与闪存子系统“INT_FLASH”。该函数提供了一种方便，可通过调用 FlashIntDisable（），在参数 ui32IntFlags 中传递 FLASH_INT_EEPROM 来禁止 EEPROM 中断
返回	无

表 H-50　EEPROMIntEnable

功能	使能 EEPROM 中断
函数原型	void　EEPROMIntEnable（uint32_t ui32IntFlags）
参数	ui32IntFlags　指示待使能的 EEPROM 中断源，当前必须为 EEPROM_INT_PROGRAM
描述	该函数使能 EEPROM 中断。使能时，在任何 EEPROM 写或擦除操作完成时会产生一个中断
返回	无

表 H-51　EEPROMIntStatus

功能	EEPROM 中断状态的报告
函数原型	uint32_tEEPROMIntStatus（bool bMasked）
参数	bMasked　确定是否屏蔽返回的中断状态。如果 bMasked 为真，返回屏蔽状态，否则返回非屏蔽状态
描述	该函数允许应用程序来查询 EEPROM 的中断状态。如果为激活状态，可通过调用 EEPROMIntClear（）来清除该中断
返回	如果发出一个中断信号返回 EEPROM_INT_PROGRAM，否则返回 0

表 H-52　EEPROMMassErase

功能	擦除 EEPROM 并返回到出厂的默认状况
函数原型	uint32_tEEPROMMassErase（void）

（续）

描述	该函数彻底擦除 EEPROM 包括删除所有块的访问保护，保留设备出厂时的默认状态。完成该操作后，所有的 EEPROM 字由值 0xFFFFFFFF 构成，以及全部的块都可以用所有 CPU 模式进行读/写操作，并且不包含密码 该函数与擦除操作同步，只有在擦除操作完成之后才会跳出该函数
返回	若返回 0 表示成功；返回非零值表示失败。而故障代码是以下组合的逻辑或： ➢ EEPROM_RC_INVPL ➢ EEPROM_RC_WRBUSY ➢ EEPROM_RC_NOPERM ➢ EEPROM_RC_WKCOPY ➢ EEPROM_RC_WKERASE ➢ EEPROM_RC_WORKING

表 H-53　EEPROMProgram

功能	写入数据到 EEPROM 中
函数原型	uint32_tEEPROMProgram（uint32_t *pui32Data, uint32_t ui32Address, uint32_t ui32Count）
参数	pui32Data　指向待写入到 EEPROM 数据第一个字的指针 ui32Address　定义待写入到 EEPROM 数据的字节地址，且这个值必须是 4 的倍数 ui32Count　定义该写入数据的字节数，且这个值必须是 4 的倍数
描述	该函数可用于在一个给定的字对齐地址将数据写入到 EEPROM 中 该函数与擦除操作同步，只有在擦除操作完成之后才会跳出该函数
返回	若返回 0 表示成功；返回非零值表示失败。而故障代码是以下组合的逻辑或： ➢ EEPROM_RC_INVPL ➢ EEPROM_RC_WRBUSY ➢ EEPROM_RC_NOPERM ➢ EEPROM_RC_WKCOPY ➢ EEPROM_RC_WKERASE ➢ EEPROM_RC_WORKING

表 H-54　EEPROMProgramNonBlocking

功能	写一个字到 EEPROM 中
函数原型	uint32_tEEPROMProgramNonBlocking（uint32_t ui32Data, uint32_t ui32Address）
参数	ui32Data　待写入到 EEPROM 中的字 ui32Address　定义待写入到 EEPROM 中数据的字节地址，且这个值必须是 4 的倍数
描述	该函数用于在 EEPROM 中断控制下编程。调用它可开始在给定的字对齐地址写一个字的数据到 EEPROM 中的进程。这个调用是异步的，且立即返回，无需等待写入完成。完成上述操作将通过 EEPROM 模块发出一根中断信号。EEPROM 外设共享一个单一的中断向量与闪存子系统“INT_FLASH”
返回	使用下列组合逻辑或的形式，返回状态和错误信息： ➢ EEPROM_RC_INVPL ➢ EEPROM_RC_WRBUSY ➢ EEPROM_RC_NOPERM ➢ EEPROM_RC_WKCOPY
返回	➢ EEPROM_RC_WKERASE ➢ EEPROM_RC_WORKING 以下标志预示操作正常，但不显示错误： ➢ EEPROM_RC_WKCOPY ➢ EEPROM_RC_WKERASE ➢ EEPROM_RC_WORKING

表 H-55　EEPROMRead

功能	从 EEPROM 中读取数据
函数原型	voidEEPROMRead（uint32_t ＊pui32Data, uint32_t ui32Address, uint32_t ui32Count）
参数	pui32Data　指向从 EEPROM 中读出数据存储单元的指针。该指针必须指向可用内存的至少 ui32Count 字节 ui32Address　从 EEPROM 中读取数据的字节地址，且这个值必须是 4 的倍数 ui32Count　从 EEPROM 中读取数据的字节数，且这个值必须是 4 的倍数
描述	该函数用于在 EEPROM 中从字对齐地址读取若干个字的数据。并由参数 pui32Data 指向复制到缓冲区的读取数据
返回	无

表 H-56　EEPROMSizeGet

功能	确定 EEPROM 的大小
函数原型	uint32_tEEPROMSizeGet（void）
描述	该函数返回以字节为单位的 EEPROM 大小
返回	返回 EEPROM 中字节的总数

表 H-57　EEPROMStatusGet

功能	返回最后 EEPROM 编程或擦除操作的状态
函数原型	uint32_tEEPROMStatusGet（void）
描述	该函数返回由 EEPROM 执行的最后编程或擦除操作的当前状态。其目的是提供应用程序编程的错误信息，或在中断控制下设置 EEPROM 的保护选项
返回	若完成最后的编程或擦除操作没有任何错误，则返回值为 0。如果操作正在进行或发生错误，将返回下列值组合的逻辑或： ➢ EEPROM_RC_INVPL ➢ EEPROM_RC_WRBUSY ➢ EEPROM_RC_NOPERM ➢ EEPROM_RC_WKCOPY ➢ EEPROM_RC_WKERASE ➢ EEPROM_RC_WORKING



附录 I　第 11 章附录：GPTM 固件库函数简介

本章附录将介绍通用定时器的固件库函数。

表 I-1　TimerADCEventGet

功能	返回可引起 ADC 触发事件的事件
函数原型	uint32_t　TimerADCEventGet（uint32_t ui32Base）
参数	ui32Base　定时器模块的基地址

（续）

描述	该函数返回可引起一个 ADC 触发事件的定时器事件。ADC 触发事件为下列任何值的逻辑或： ➢ TIMER_ADC_MODEMATCH_B //已使能定时器 B 的模式匹配 ADC 触发 ➢ TIMER_ADC_CAPEVENT_B //已使能定时器 B 的捕获事件 ADC 触发 ➢ TIMER_ADC_CAPMATCH_B //已使能定时器 B 的捕获匹配 ADC 触发 ➢ TIMER_ADC_TIMEOUT_B //已使能定时器 B 的超时 ADC 触发 ➢ TIMER_ADC_MODEMATCH_A //已使能定时器 A 的模式匹配 ADC 触发 ➢ TIMER_ADC_RTC_A // 已使能定时器 A 的 RTC ADC 触发 ➢ TIMER_ADC_CAPEVENT_A //已使能定时器 A 的捕获事件 ADC 触发 ➢ TIMER_ADC_CAPMATCH_A //已使能定时器 A 的捕获匹配 ADC 触发 ➢ TIMER_ADC_TIMEOUT_A //已使能定时器 A 的超时 ADC 触发
返回值	定时器触发 ADC 的事件

表 I-2　TimerADCEventSet

功能	使能引起 ADC 触发事件的事件
函数原型	void　TimerADCEventSet（uint32_t ui32Base，uint32_t ui32ADCEvent）
参数	ui32Base　定时器模块的基地址 ui32ADCEvent　引起 ADC 触发事件的位屏蔽
描述	该函数使能引起定时器 ADC 触发事件的事件。而 ADC 触发事件可通过下列任意值的逻辑或运算得到的参数 ui32ADCEvent 指定： ➢ TIMER_ADC_MODEMATCH_B //使能定时器 B 的模式匹配 ADC 触发 ➢ TIMER_ADC_CAPEVENT_B //使能定时器 B 的捕获事件 ADC 触发 ➢ TIMER_ADC_CAPMATCH_B //使能定时器 B 的捕获匹配 ADC 触发 ➢ TIMER_ADC_TIMEOUT_B //使能定时器 B 的超时 ADC 触发 ➢ TIMER_ADC_MODEMATCH_A //使能定时器 A 的模式匹配 ADC 触发 ➢ TIMER_ADC_RTC_A //使能定时器 A 的 RTC ADC 触发 ➢ TIMER_ADC_CAPEVENT_A //使能定时器 A 的捕获事件 ADC 触发 ➢ TIMER_ADC_CAPMATCH_A //使能定时器 A 的捕获匹配 ADC 触发 ➢ TIMER_ADC_TIMEOUT_A //使能定时器 A 的超时 ADC 触发
返回值	无

表 I-3　TimerClockSourceGet

功能	返回指定定时器模块的时钟源
函数原型	uint32_t　TimerClockSourceGet（uint32_t ui32Base）
参数	ui32Base　定时器模块的基地址
描述	该函数返回指定的定时器模块的时钟源。候选的时钟源为系统时钟（TIMER_CLOCK_SYSTEM）或内部精密振荡器（TIMER_CLOCK_PIOSC）
返回值	要么返回 TIMER_CLOCK_SYSTEM，要么返回 TIMER_CLOCK_PIOSC

表 I-4　TimerClockSourceSet

功能	设置指定的定时器模块的时钟源
函数原型	uint32_t　TimerADCEventGet（uint32_t ui32Base）
参数	ui32Base　定时器模块的基地址
描述	该函数设置给定的定时器模块中定时器 A 和定时器 B 的时钟源。候选的时钟源为系统时钟（TIMER_CLOCK_SYSTEM）或内部精密振荡器（TIMER_CLOCK_PIOSC）
返回值	无

表 I-5 TimerConfigure

功能	配置定时器
函数原型	void TimerConfigure（uint32_t ui32Base，uint32_t ui32Config）
参数	ui32Base 定时器模快的基地址 ui32Config 配置定时器
描述	该函数配置定时器的的操作模式。定时器模块在配置前被禁止并保持禁止状态。定时器可用 TIMER_CFG_＊值来配置成一个单一的全宽定时器，或使用 TIMER_CFG_A_＊和 TIMER_CFG_B_＊值来配置成一对半宽的定时器来传递参数 ui32Config ui32Config 指定的配置为下列值中的一个： ➢ TIMER_CFG_ONE_SHOT //全宽单次触发定时器 ➢ TIMER_CFG_ONE_SHOT_UP //全宽单次递增替代递减计数定时器（不是所有的器件都提供） ➢ TIMER_CFG_PERIODIC //全宽周期定时器 ➢ TIMER_CFG_PERIODIC_UP //全宽周期递增替代递减计数定时器（不是所有的器件都提供） ➢ TIMER_CFG_RTC //全宽实时时钟定时器 ➢ TIMER_CFG_SPLIT_PAIR // 2 个半宽定时器 当配置成一对半宽定时器时，每个定时器可单独配置。第一个定时器通过下列的一个值和 ui32Config 的逻辑或操作结果来配置 ➢ TIMER_CFG_A_ONE_SHOT //半宽单次触发定时器 ➢ TIMER_CFG_A_ONE_SHOT_UP //半宽单次递增替代递减计数定时器（不是所有的部件都提供） ➢ TIMER_CFG_A_PERIODIC //半宽周期定时器 ➢ TIMER_CFG_A_PERIODIC_UP //半宽周期递增替代递减计数定时器（不是所有的部件都提供） ➢ TIMER_CFG_A_CAP_COUNT //半宽边沿计数捕获 ➢ TIMER_CFG_A_CAP_COUNT_UP //半宽边沿递增替代递减计数捕获（不是所有的部件都提供） ➢ TIMER_CFG_A_CAP_TIME //半宽边沿时间捕获 ➢ TIMER_CFG_A_CAP_TIME_UP //半宽边沿递增替代递减定时捕获（不是所有的部件都提供） ➢ TIMER_CFG_A_PWM //半宽 PWM 输出 类似的，第二个定时器是通过将 ui32Config 设置为一个相应的 TIMER_CFG_B_＊值和 ui32Config 间的逻辑或操作结果来配置
返回值	无

表 I-6 TimerControlEvent

功能	控制事件类型
函数原型	void TimerControlEvent（uint32_t ui32Base, uint32_t ui32Timer, uint32_t ui32Event）
参数	ui32Base 定时器模块的基地址 ui32Timer 指定待调整的定时器，它必须为 TIMER_A、TIMER_B 或 TIMER_BOTH 其中之一 ui32Event 指定事件的类型，必须为 TIMER_EVENT_POS_EDGE、TIMER_EVENT_NEG_EDG 或 TIMER_EVENT_BOTH_EDGES 中的一个
描述	该函数设置在捕获模式中触发定时器的信号沿
返回值	无

表 I-7　TimerControlLevel

功能	控制输出电平
函数原型	void　TimerControlLevel（uint32_t ui32Base, uint32_t ui32Timer, bool bInvert）
参数	ui32Base　定时器模块的基地址 ui32Timer　指定待调整的定时器，它必须为 TIMER_A、TIMER_B 或 TIMER_BOTH 其中之一 bInvert　指定输出电平
描述	该函数用于设置指定定时器的 PWM 输出电平。若参数 bInvert 为 true，定时器输出低电平有效；否则，定时器输出为高电平
返回值	无

表 I-8　TimerControlStall

功能	控制停止处理
函数原型	void　TimerControlStall（uint32_t ui32Base, uint32_t ui32Timer, bool bStall）
参数	ui32Base　定时器模块的基地址 ui32Timer　指定待调整的定时器，必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 其中之一 bStall　指定中止信号的响应
描述	该函数控制指定的定时器的停止响应。若参数 bStall 为 true，定时器在处理器进入调试模式时将停止计数；否则，定时器在调试模式中将继续运行
返回值	无

表 I-9　TimerControlTrigger

功能	使能或禁止 ADC 触发输出
函数原型	void TimerControlTrigger（uint32_t ui32Base, uint32_t ui32Timer, bool bEnable）
参数	ui32Base　定时器模块的基地址 ui32Timer　指定待调整的定时器，必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 其中之一 bEnable　指定 ADC 所需的触发状态
描述	该函数控制指定的定时器的 ADC 触发输出。若参数 bEnable 为 true，使能定时器输出触发；否则，禁止定时器触发输出
返回值	无

表 I-10　TimerControlWaitOnTrigger

功能	控制触发处理等待
函数原型	void　TimerControlWaitOnTrigger（uint32_t ui32Base, uint32_t ui32Timer, bool bWait）
参数	ui32Base　定时器模快的基地址 ui32Timer　指定待调整的定时器，必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 其中之一 bWait　指定是否应该等待定时器触发输入
描述	该函数控制定时器在开始计数时是否需等待一个触发输入。使能时，在触发链之前的定时器必须计数到超时，以便这个定时器开始计数。详细资料请参照技术手册中有关触发器链的描述
返回值	无

表 I-11　TimerDisable

功能	禁止定时器
函数原型	void　TimerDisable（uint32_t　ui32Base， uint32_t　ui32Timer）
参数	ui32Base　定时器模块的基地址 ui32Timer　指定要调整的定时器，且必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 其中之一
描述	该函数用于禁止定时器模块操作
返回值	无

表 I-12　TimerDMAEventGe

功能	返回可触发 DMA 请求的事件
函数原型	uint32_t　TimerDMAEventGet（uint32_t ui32Base）
参数	ui32Base　定时器模块的基地址
描述	该函数返回定时器能触发 DMA 序列开始的事件。而 DMA 触发事件为下列值的逻辑或： ➢ TIMER_ADC_MODEMATCH_B　//使能定时器 B 的模式匹配 DMA 触发 ➢ TIMER_ADC_CAPEVENT_B　//使能定时器 B 的捕获事件 DMA 触发 ➢ TIMER_ADC_CAPMATCH_B　//使能定时器 B 的捕获匹配 DMA 触发 ➢ TIMER_ADC_TIMEOUT_B　//使能定时器 B 的超时 DMA 触发 ➢ TIMER_ADC_MODEMATCH_A　//使能定时器 A 的模式匹配 DMA 触发 ➢ TIMER_ADC_RTC_A　//使能定时器 A 的 RTC DAM 触发 ➢ TIMER_ADC_CAPEVENT_A　//使能定时器 A 的捕获事件 DMA 触发 ➢ TIMER_ADC_CAPMATCH_A　//使能定时器 A 的捕获匹配 DMA 触发 ➢ TIMER_ADC_TIMEOUT_A　//使能定时器 A 的超时 DMA 触发
返回值	定时器触发 uDMA 的事件

表 I-13　TimerDMAEventSet

功能	禁止定时器
函数原型	void　TimerDMAEventSet（uint32_t ui32Base，uint32_t ui32DMAEvent）
参数	ui32Base　定时器模块的基地址 ui32DMAEvent　可触发 DMA 事件的位屏蔽
描述	该函数使能定时器可触发 DMA 序列开始的事件。而 D MA 触发事件可在参数 ui32DMAEvent 中指定，该值可从下列值的逻辑或运算的结果中得到： ➢ TIMER_ADC_MODEMATCH_B　//已使能定时器 B 的模式匹配 ADC 触发 ➢ TIMER_ADC_CAPEVENT_B　//已使能定时器 B 的捕获事件 ADC 触发 ➢ TIMER_ADC_CAPMATCH_B　//已使能定时器 B 的捕获匹配 ADC 触发 ➢ TIMER_ADC_TIMEOUT_B　//已使能定时器 B 的超时 ADC 触发 ➢ TIMER_ADC_MODEMATCH_A　//已使能定时器 A 的模式匹配 ADC 触发 ➢ TIMER_ADC_RTC_A　// 已使能定时器 A 的 RTC ADC 触发 ➢ TIMER_ADC_CAPEVENT_A　//已使能定时器 A 的捕获事件 ADC 触发 ➢ TIMER_ADC_CAPMATCH_A　//已使能定时器 A 的捕获匹配 ADC 触发 ➢ TIMER_ADC_TIMEOUT_A　//已使能定时器 A 的超时 ADC 触发
返回值	无

表 I-14　TimerEnable

功能	使能定时器
函数原型	void TimerEnable（uint32_t ui32Base，uint32_t ui32Timer）
参数	ui32Base　定时器模块的基地址 ui32Timer　指定待调整的定时器，必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 其中之一
描述	该函数使能定时器模块的操作，并且该定时器必须在使能前配置
返回值	无

表 I-15 TimerIntClear

功能	清除定时器的中断源
函数原型	void TimerIntClear（uint32_t ui32Base，uint32_t ui32IntFlags）
参数	ui32Base 定时器模块的基地址 ui32IntFlags 待清除中断源的位屏蔽
描述	指定待清除定时器的中断源，使之不再有效。该函数必须在中断处理程序中调用，以免在退出时立即再次对其触发 参数 ui32IntFlags 和 TimerIntEnable( ) 中的参数 ui32IntFlags 具有相同的定义
返回值	无

表 I-16 TimerIntDisable

功能	禁止单个定时器的中断源
函数原型	void TimerIntDisable（uint32_t ui32Base，uint32_t ui32IntFlags）
参数	ui32Base 定时器模块的基地址 ui32IntFlags 待禁止中断源的位屏蔽
描述	该函数禁止指定定时器的中断源
返回值	无

表 I-17 TimerIntEnable

功能	使能单个定时器的中断源
函数原型	void TimerIntEnable（uint32_t ui32Base，uint32_t ui32IntFlags）
参数	ui32Base 定时器模块的基地址 ui32IntFlags 待使能中断源的位屏蔽
描述	该函数使能指定定时器的中断源 参数 ui32IntFlags 必须为以下任意组合的逻辑或： ➢ TIMER_TIMB_DMA //定时器 B DMA 完成 ➢ TIMER_TIMA_DMA //定时器 A DMA 完成 ➢ TIMER_CAPB_EVENT //捕获 B 事件中断 ➢ TIMER_CAPB_MATCH //捕获 B 匹配中断 ➢ TIMER_TIMB_TIMEOUT //定时器 B 超时中断 ➢ TIMER_RTC_MATCH //RTC 中断屏蔽 ➢ TIMER_CAPA_EVENT //捕获 A 事件中断 ➢ TIMER_CAPA_MATCH //捕获 A 匹配中断 ➢ TIMER_TIMA_TIMEOUT //定时器 A 超时中断
返回值	无

表 I-18 TimerIntRegister

功能	注册定时器中断的中断处理程序
函数原型	void TimerIntRegister（uint32_t ui32Base, uint32_t ui32Timer, void（ * pfnHandler）（void））
参数	ui32Base 定时器模块的基地址 ui32Timer 指定定时器，必须为 TIMER_A、TIMER_B 或 TIMER_BOTH 其中之一 pfnHandler 指向发生定时器中断时被调函数的指针
描述	该函数用于注册在发生定时器中断时需调用的处理程序。此外，该函数可使能中断控制器中的全局中断，特定定时器中断必须通过 TimerIntEnable( ) 函数使能，而调用 TimerInt-Clear( ) 来清除中断源是中断处理程序的责任
返回值	无

表 I-19　TimerIntStatus

功能	获取当前的中断状态
函数原型	uint32_t　TimerIntStatus（uint32_t　ui32Base, bool bMasked）
参数	ui32Base　定时器模块的基地址 bMasked　若要求原始中断状态，bMasked 取 false；若要求中断屏蔽状态，bMasked 取 true
描述	该函数返回定时器模块的中断状态。无论原始中断状态，还是允许反映到处理器的中断状态都会被返回
返回值	当前的中断状态，由 TimerIntEnable( ) 来枚举位字段的值

表 I-20　TimerIntUnregister

功能	注销定时器中断的中断处理程序
函数原型	void　TimerIntUnregister（uint32_t ui32Base, uint32_t ui32Timer）
参数	ui32Base　定时器模块的基地址 ui32Timer　指定定时器，必须为 TIMER_A、TIMER_B 或 TIMER_BOTH 其中之一
描述	该函数用于注销在发生定时器中断时需调用的处理程序。这个函数也将屏蔽中断控制器中的中断，以便中断处理程序不再被调用
返回值	无

表 I-21　TimerLoadGet

功能	获取定时器的装载值
函数原型	uint32_t　TimerLoadGet（uint32_t ui32Base, uint32_t ui32Timer）
参数	ui32Base　定时器模块的基地址 ui32Timer　指定定时器，必须为 TIMER_A 或 TIMER_B 其中之一。当定时器被配置成全宽操作时，只有 TIMER_A 可用
描述	该函数用于获取指定定时器当前可编程间隙的装载值
返回值	返回定时器的装载值

表 I-22　TimerLoadGet64

功能	获取 64 位定时器的装载值
函数原型	uint64_t　TimerLoadGet64（uint32_t ui32Base）
参数	ui32Base　定时器模块的基地址
描述	该函数用于获取指定 64 位定时器的当前可编程间隙的装载值
返回值	返回定时器的装载值

表 I-23　TimerLoadSet

功能	设置定时器装载值
函数原型	void　TimerLoadSet（uint32_t ui32Base, uint32_t ui32Timer, uint32_t ui32Value）

（续）

参数	ui32Base　定时器模块的基地址 ui32Timer　指定定时器，必须为 TIMER_A、TIMER_B 或 TIMER_BOTH 其中之一。当定时器配置为全宽操作时，只有 TIMER_A 可用 ui32Value　装载值
描述	该函数配置定时器的装载值；若定时器正在运行，该值将被立即加载到定时器中
返回值	无

表 I-24　TimerLoadSet64

功能	设置 64 位定时器的装载值
函数原型	void TimerLoadSet64（uint32_t ui32Base, uint64_t ui64Value）
参数	ui32Base　定时器模块的基地址 ui64Value　装载值
描述	该函数配置 64 位定时器的装载值；若定时器正在运行，该值将被立即加载到定时器中
返回值	无

表 I-25　TimerMatchGet

功能	获取定时器的匹配值
函数原型	uint32_t　TimerMatchGet（uint32_t ui32Base, uint32_t ui32Timer）
参数	ui32Base　定时器模块的基地址 ui32Timer　指定定时器，必须为 TIMER_A、TIMER_B 其中之一。当定时器被配置成全宽操作时，只有 TIMER_A 可用
描述	该函数获取指定定时器的匹配值
返回值	返回定时器的匹配值

表 I-26　TimerMatchGet64

功能	获取 64 位定时器的匹配值
函数原型	uint64_t　TimerMatchGet64（uint32_t ui32Base）
参数	ui32Base　定时器模块的基地址
描述	该函数获取指定定时器的匹配值
返回值	返回定时器的匹配值

表 I-27　TimerMatchSet

功能	设置定时器的匹配值
函数原型	void　TimerMatchSet（uint32_t ui32Base, uint32_t ui32Timer, uint32_t ui32Value）
参数	ui32Base　定时器模块的基地址 ui32Timer　指定定时器，必须为 TIMER_A、TIMER_B 或 TIMER_BOTH 其中之一。当定时器配置为全宽操作时，只有 TIMER_A 可用 ui32Value　匹配值
描述	该函数配置定时器的匹配值。该值用于在捕获计数模式中决定什么时候中断处理器，以及在 PWM 模式中用它来确定输出信号的占空比。在一些 Tiva 器件中，匹配中断也可以在周期和单触发模式中生成
返回值	无

表 I-28　TimerMatchSet64

功能	设置 64 位定时器的匹配值
函数原型	void　TimerMatchSet64（uint32_t ui32Base, uint64_t ui64Value）
参数	ui32Base　定时器模块的基地址 ui64Value　匹配值
描述	该函数配置定时器的匹配值。该值用于在捕获计数模式中决定什么时候中断处理器，以及在 PWM 模式中用它来确定输出信号的占空比
返回值	无

表 I-29　TimerPrescaleGet

功能	获取定时器的预分频值
函数原型	uint32_t　TimerPrescaleGet（uint32_t ui32Base, uint32_t ui32Timer）
参数	ui32Base　定时器模块的基地址 ui32Timer　指定定时器，必须为 TIMER_A、TIMER_B 其中之一
描述	该函数获取输入时钟预分频器的值。预分频器只在半宽的模式中操作，并且可用于扩展半宽定时器模式的范围。预分频器为处于周期和单次模式的递减计数提供了最低有效位；而对于其他所有模式，预分频器却提供最高有效位
返回值	定时器预分频器的值

表 I-30　TimerPrescaleMatchGet

功能	获取定时器预分频的匹配值
函数原型	uint32_t　TimerPrescaleMatchGet（uint32_t ui32Base, uint32_t ui32Timer）
参数	ui32Base　定时器模块的基地址 ui32Timer　指定定时器，必须为 TIMER_A、TIMER_B 其中之一
描述	该函数获取输入时钟预分频器的匹配值。在使用计数器匹配和预分频器的半宽模式中，预分频匹配会有效的扩展匹配的范围
返回值	定时器预分频的匹配值

表 I-31　TimerPrescaleMatchSet

功能	设置定时器预分频的匹配值
函数原型	void TimerPrescaleMatchSet（uint32_t ui32Base, uint32_t ui32Timer, uint32_t ui32Value）
参数	ui32Base　定时器模块的基地址 ui32Timer　指定定时器，必须为 TIMER_A、TIMER_B 或 TIMER_BOTH 其中之一 ui32Value　定时器预分频匹配值。对于 16/32 位定时器，该值必须在 0～255 之间（包括边界）；对于 32/64 位定时器，该值必须在 0～65535 之间（包括边界）
描述	该函数配置输入时钟预分频器的匹配值
返回值	无

表 I-32　TimerPrescaleSet

功能	设置定时器的预分频值
函数原型	void　TimerPrescaleSet（uint32_t ui32Base, uint32_t ui32Timer, uint32_t ui32Value）

参数	ui32Base　定时器模块的基地址 ui32Timer　指定定时器，必须为 TIMER_A、TIMER_B 或 TIMER_BOTH 其中之一 ui32Value　定时器预分频匹配值
描述	该函数配置输入时钟预分频器的值
返回值	无

表 I-33　TimerRTCDisable

功能	禁止 RTC 计数
函数原型	void　TimerRTCDisable（uint32_t ui32Base）
参数	ui32Base　定时器模块的基地址
描述	该函数用于在 RTC 模式中停止定时器计数
返回值	无

表 I-34　TimerRTCEnable

功能	使能 RTC 计数
函数原型	void　TimerRTCEnable（uint32_t ui32Base）
参数	ui32Base　定时器模快的基地址
描述	该函数用于在 RTC 模式中使定时器开始计数。如果定时器未被配置成 RTC 模式，则该函数无效
返回值	无

表 I-35　TimerSynchronize

功能	一组定时器中的同步计数器
函数原型	void　TimerSynchronize（uint32_t ui32Base, uint32_t ui32Timers）
参数	ui32Base　定时器模块的基地址，该参数必须为 Timer0 的基址（即 TIMER0_BASE） ui32Timer　设置定时器同步
描述	该函数用于在一组指定定时器中同步计数器。当定时器在半宽模式中运行时，每一半可被包括或排除在该同步事件。当定时器运行在全宽模式中，仅定时器 A 可被同步（指定的定时器 B 没有作用） 参数 ui32Timers 可以是下列定义的任何组合的逻辑或： ➢ TIMER_0A_SYNC ➢ TIMER_0B_SYNC ➢ TIMER_1A_SYN ➢ TIMER_1B_SYNC ➢ TIMER_2A_SYNC ➢ TIMER_2B_SYNC ➢ TIMER_3A_SYNC ➢ TIMER_3B_SYNC ➢ TIMER_4A_SYNC ➢ TIMER_4B_SYNC ➢ TIMER_5A_SYNC ➢ TIMER_5B_SYNC ➢ WTIMER_0A_SYNC ➢ WTIMER_0B_SYNC ➢ WTIMER_1A_SYNC ➢ WTIMER_1B_SYNC ➢ WTIMER_2A_SYNC ➢ WTIMER_2B_SYNC ➢ WTIMER_3A_SYNC ➢ WTIMER_3B_SYNC ➢ WTIMER_4A_SYNC

（续）

描述	➢ WTIMER_4B_SYNC ➢ WTIMER_5A_SYNC ➢ WTIMER_5B_SYNC
返回值	无

表 I-36　TimerValueGet

功能	获取当前定时器的值
函数原型	uint32_t　TimerValueGet（uint32_t ui32Base, uint32_t ui32Timer）
参数	ui32Base　定时器模块的基地址 ui32Timer　指定定时器，必须为 TIMER_A、TIMER_B 其中之一。当定时器配置成全宽的操作时，只能使用 TIMER_A
描述	该函数读取指定定时器的当前值
返回值	返回定时器的当前值

表 I-37　TimerValueGet64

功能	获取 64 位定时器当前值
函数原型	uint64_t　TimerValueGet64（uint32_t ui32Base）
参数	ui32Base　定时器模块的基地址
描述	该函数读取指定定时器的当前值
返回值	返回定时器的当前值



附录 J　第 12 章附录：PWM 固件库函数简介

本章附录将介绍 Tiva C 系列芯片的 PWM API 库函数。

表 J-1　PWMClockGet

功能	获取当前 PWM 的时钟配置
函数原型	uint32_t　PWMClockGet（uint32_t ui32Base）
参数	ui32Base　PWM 模块的基地址
描述	该函数返回当前 PWM 的时钟配置
返回参数	返回当前 PWM 的时钟配置，必须为下列值之一： ➢ PWM_SYSCLK_DIV_1 ➢ PWM_SYSCLK_DIV_2 ➢ PWM_SYSCLK_DIV_4 ➢ PWM_SYSCLK_DIV_8 ➢ PWM_SYSCLK_DIV_16 ➢ PWM_SYSCLK_DIV_32 ➢ PWM_SYSCLK_DIV_64

表 J-2 PWMClockSet

功能	设置当前 PWM 的时钟配置
函数原型	void PWMClockSet（uint32_t ui32Base, uint32_t ui32Config）
参数	ui32Base PWM 模块的基地址 ui32Config PWM 的时钟配置，必须为下列值之一： ➢ PWM_SYSCLK_DIV_1 ➢ PWM_SYSCLK_DIV_2 ➢ PWM_SYSCLK_DIV_4 ➢ PWM_SYSCLK_DIV_8 ➢ PWM_SYSCLK_DIV_16 ➢ PWM_SYSCLK_DIV_32 ➢ PWM_SYSCLK_DIV_64
描述	该函数设置 PWM 的时钟分频器作为 PWM 时钟源。它还配置 PWM 模块的时钟频率作为系统时钟的分频。这个时钟用于 PWM 模块产生 PWM 信号；它的速度构成所有 PWM 信号的基础
返回参数	无

表 J-3 PWMDeadBandDisable

功能	禁止 PWM 死区输出
函数原型	void PWMDeadBandDisable（uint32_t ui32Base, uint32_t ui32Gen）
参数	ui32Base PWM 模块的基地址 ui32Gen 修改 PWM 发生器，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3
描述	该函数禁止指定 PWM 发生器的死区模式，可将 OUTA 和 OUTB 信号去耦
返回参数	无

表 J-4 PWMDeadBandEnable

功能	使能 PWM 死区输出与设置死区延迟
函数原型	void PWMDeadBandEnable（uint32_t ui32Base, uint32_t ui32Gen, uint16_t ui16Rise, uint16_t ui16Fall）
参数	ui32Base PWM 模块的基地址 ui32Gen 修改 PWM 发生器，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 ui16Rise 指定上升沿的延迟宽度 ui16Fall 指定下降沿的延迟宽度
描述	该函数设置指定 PWM 发生器的死区，这里死区被定义为从 PWM 发生器 OUTA 信号的上升沿或下降沿开始的 PWM 时钟节拍数。注意，这个函数导致 OUTB 耦合到 OUTA
返回参数	无

表 J-5　PWMFaultIntClear

功能	清除 PWM 模块的故障中断
函数原型	void PWMFaultIntClear（uint32_t ui32Base）
参数	ui32Base　PWM 模块的基地址
描述	该函数通过在所选 PWM 模块的中断状态寄存器中写入相应位来清除故障中断。此函数仅能清除 FAULT0 中断并保留向后兼容。推荐采用 PWMFaultIntClearExt（）替代 PWM-FaultIntClear( )函数，因为它支持设备上给出的所有故障中断，且无论是否带有扩展 PWM 故障处理程序均能得到支持
返回参数	无

表 J-6　PWMFaultIntClearExt

功能	清除 PWM 模块的故障中断
函数原型	void PWMFaultIntClearExt（uint32_t ui32Base, uint32_t ui32FaultInts）
参数	ui32Base　PWM 模块的基地址 ui32FaultInts　指定待清除的故障中断
描述	该函数通过在所选 PWM 模块的 PWM 中断状态寄存器中写入相应位来清除一个或多个故障中断 参数 ui32FaultInts 必须为下列值的逻辑或： ➢ PWM_INT_FAULT0 ➢ PWM_INT_FAULT1 ➢ PWM_INT_FAULT2 ➢ PWM_INT_FAULT3 在设备上运行支持扩展 PWM 故障的处理程序时，通过执行给定生成器每个所配置的故障触发信号的逻辑或结果来驱动故障中断。故这些中断不是直接与设备候选的 4 个 FAULTn 输入关联，但这表明发出的故障信号已传达到了 4 个候选的 PWM 发生器的其中之一。对于没有扩展 PWM 故障处理程序的设备，则中断与单一故障引脚的状态直接关联
返回参数	无

表 J-7　PWMFaultIntRegister

功能	注册 PWM 模块中的故障状态检测的处理处理程序
函数原型	void　PWMFaultIntRegister（uint32_t ui32Base，void（ * pfnIntHandler）（void））
参数	ui32Base　PWM 模块的基地址 pfnIntHandler　指向当 PWM 故障中断发生时被调函数的指针
描述	当检测到所选 PWM 模块中的故障中断时，该函数可确保由 pfnIntHandler 指定的中断处理程序被调用。该函数还应该在 NVIC 中使能 PWM 故障中断，以及必须通过 PWMIntEna-ble（）在模块级使能 PWM 故障中断
返回参数	无

表 J-8　PWMFaultIntUnregister

功能	注销 PWM 故障状态的中断处理程序
函数原型	void PWMFaultIntUnregister（uint32_t ui32Base）
参数	ui32Base　PWM 模块的基地址
描述	该函数从所选 PWM 模块中删除 PWM 故障中断的中断处理程序。该函数还应该在 NVIC 中使能 PWM 故障中断，以及必须通过 PWMIntEnable（）在模块级使能 PWM 故障中断
返回参数	无

表 J-9 PWMGenConfigure

功能	配置 PWM 发生器
函数原型	void PWMGenConfigure（uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Config）
参数	ui32Base PWM 模块的基地址 ui32Gen 修改 PWM 发生器，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 ui32Config PWM 发生器的配置
描述	该函数用于设置 PWM 发生器操作模式。可配置成计数模式、同步模式，以及调试行为。完成配置后发生器将保持禁止状态 PWM 发生器具有两种不同的计数模式：递减计数模式或先递增后递减计数模式。在递减计数模式时，将从装载值递减计数到零，然后重置为装载值，可生成左对齐的 PWM 信号。在先递增后递减计数模式下，将先从零递增计数到转载值，然后递减计数到零，然后重复该过程，可生成中心对齐的 PWM 信号。若修改 PWM 发生器的参数（周期和脉冲宽度），其对 PWM 信号输出的影响可被延迟。在同步模式下，只有发生同步事件后才可进行参数更新。这种模式允许对多个参数进行修改并同时生效，而不是一次一个。此外，在同步模式下，多个 PWM 发生器参数可以同时更新，允许其被当作一个统一发生器来处理。在非同步模式中，参数的更新不会延时到同步事件的发生。在两种模式下，仅在计数器的值为零时，才会发生参数更新，以防止在参数更新过程中形成奇异的 PWM 信号 在处理器因调试器停止时，此时，PWM 发生器既可以暂停，也可继续运行。如果配置成暂停，它将继续计数直到零为止，在该点发生器将被暂停，直到处理器重新启动。如果配置成连续运行，PWM 发生器将会一直计数 参数 ui32Config 包含所需的配置，为以下值的逻辑或： ➢ PWM_GEN_MODE_DOWN 或 PWM_GEN_MODE_UP_DOWN//指定计数模式 ➢ PWM_GEN_MODE_SYNC 或 PWM_GEN_MODE_NO_SYNC//指定计数器的装载值和比较器更新同步模式 ➢ PWM_GEN_MODE_DBG_RUN 或 PWM_GEN_MODE_DBG_STOP//指定调试的行为 ➢ PWM_GEN_MODE_GEN_NO_SYNC、PWM_GEN_MODE_GEN_SYNC_LOCAL 或 PWM_GEN_MODE_GEN_SYNC_GLOBAL//指定发生器计数模式的变化同步更新模式 ➢ PWM_GEN_MODE_DB_NO_SYNC、PWM_GEN_MODE_DB_SYNC_LOCAL 或 PWM_GEN_MODE_DB_SYNC_GLOBAL//指定死区参数的同步模式 ➢ PWM_GEN_MODE_FAULT_LATCHED 或 PWM_GEN_MODE_FAULT_UNLATCHED//指定是否锁定或故障状态 ➢ PWM_GEN_MODE_FAULT_MINPER 或 PWM_GEN_MODE_FAULT_NO_MINPER//指定是否需要最小故障周期支持 ➢ PWM_GEN_MODE_FAULT_EXT 或 PWM_GEN_MODE_FAULT_LEGACY//指定是否使能扩展故障源选择支持 设置 PWM_GEN_MODE_FAULT_MINPER 允许应用程序设置 PWM 故障信号的最小持续时间。即使外部故障引脚已经取消了故障状态，故障信号也至少会产生一次。使用这种模式时应该小心，因为 PWM 产生的故障中断是一直有效的。因此，如果中断处理函数的退出早于故障定时器所规定的时间，有可能会立即再次触发中断处理程序
返回参数	无

表 J-10 PWMGenDisable

功能	禁止 PWM 发生器模块的定时器/计数器
函数原型	void PWMGenDisable（uint32_t ui32Base, uint32_t ui32Gen）

（续）

参数	ui32Base　PWM 模块的基地址 ui32Gen　禁止 PWM 发生器，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3
描述	该函数封锁 PWM 时钟驱动指定发生器模块的定时器/计数器
返回参数	无

表 J-11　PWMGenEnable

功能	使能 PWM 发生器模块的定时器/计数器
函数原型	void PWMGenEnable（uint32_t ui32Base, uint32_t ui32Gen）
参数	ui32Base　PWM 模块的基地址 ui32Gen　禁止 PWM 发生器，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3
描述	该函数允许 PWM 时钟驱动指定的发生器的定时器/计数器
返回参数	无

表 J-12　PWMGenFaultClear

功能	清除给定 PWM 发生器的一个或多个锁存故障触发
函数原型	void PWMGenFaultClear（uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Group, uint32_t ui32FaultTriggers）
参数	ui32Base　PWM 模块的基地址 ui32Gen　正在查询的 PWM 发生器故障触发状态，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 ui32Group　指示被查询故障的子集，该参数必须为 PWM_FAULT_GROUP_0 或 PWM_FAULT_GROUP_1 ui32FaultTriggers　被清除的故障触发器集
描述	该函数允许应用程序清除给定 PWM 发生器的故障触发。只有之前用参数 ui32Config 中的标志 PWM_GEN_MODE_FAULT_LATCHED 调用 PWMGenConfigure( )，才需要用该函数
返回参数	无

表 J-13　PWMGenFaultConfigure

功能	配置指定 PWM 发生器的最小故障周期与故障引脚意义（sense）
函数原型	void PWMGenFaultConfigure（uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32MinFaultPeriod, uint32_t ui32FaultSenses）

参数	ui32Base　PWM 模块的基地址 ui32Gen　PWM 发生器的故障配置设置，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 ui32MinFaultPeriod　表示在 PWM 时钟周期中的最小故障激活周期 ui32FaultSenses　指示故障输入的何种状态为有效。有效值为 PWM_FAULTn_SENSE_HIGH 与 PWM_FAULTn_SENSE_LOW 的逻辑或
描述	该函数配置给定发生器的最小故障周期与 4 个候选的故障输入中各自的意义（sense）。以 PWM 时钟周期表示最小故障周期，仅当用设置在参数 ui32Config 中的标志 PWM_GEN_MODE_FAULT_PER 来调用 PWMGenConfigure( ) 函数时，它才会生效。当故障输入有效时，最小故障周期定时器确保其至少在指定的时钟周期数内仍然有效
返回参数	无

表 J-14　PWMGenFaultStatus

功能	返回给定 PWM 发生器的当前故障触发状态
函数原型	uint32_t　PWMGenFaultStatus（uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Group）
参数	ui32Base　PWM 模块的基地址 ui32Gen　正在查询故障触发状态的 PWM 发生器，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 ui32Group　指示被查询故障的子集，该参数必须为 PWM_FAULT_GROUP_0 或 PWM_FAULT_GROUP_1
描述	该函数允许应用程序查询给定 PWM 发生器每个故障触发器输入的当前状态。返回每个故障触发输入的当前状态，除非使用参数 ui32Config 中的标志 PWM_GEN_MODE_FAULT_LATCHEDP 先期调用 PWMGenConfigure ( ) 函数，在这种情况下，返回的状态为锁存故障触发的状态 如果配置成锁存故障，应用程序必须调用 PWMGenFaultClear( ) 来清除每个触发
返回参数	返回给定的 PWM 发生器故障触发的当前状态。置位表示相关的触发被激活的 对于 PWM_FAULT_GROUP_0，返回值为下列值的逻辑或： ➢ PWM_FAULT_FAULT0 ➢ PWM_FAULT_FAULT1 ➢ PWM_FAULT_FAULT2 ➢ PWM_FAULT_FAULT3 对于 PWM_FAULT_GROUP_1，返回值为下列值的逻辑或： ➢ PWM_FAULT_DCMP0 ➢ PWM_FAULT_DCMP1 ➢ PWM_FAULT_DCMP2 ➢ PWM_FAULT_DCMP3 ➢ PWM_FAULT_DCMP4 ➢ PWM_FAULT_DCMP5 ➢ PWM_FAULT_DCMP6 ➢ PWM_FAULT_DCMP7

表 J-15　PWMGenFaultTriggerGet

功能	返回给定 PWM 发生器当前配置的故障触发设置
函数原型	uint32_t　PWMGenFaultTriggerGet（uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Group）
参数	ui32Base　PWM 模块的基地址 ui32Gen　正在查询故障触发的 PWM 发生器，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 ui32Group　指示被查询故障的子集。这个参数必须为 PWM_FAULT_GROUP_0 或 PWM_FAULT_GROUP_1
描述	该函数允许应用程序查询当前的输入设置，这有助于产生给定的 PWM 发生器的故障状态
返回参数	返回所提供故障组的当前故障触发配置 对于 PWM_FAULT_GROUP_0，返回值为下列值的逻辑或： ➢ PWM_FAULT_FAULT0 ➢ PWM_FAULT_FAULT1 ➢ PWM_FAULT_FAULT2 ➢ PWM_FAULT_FAULT3 对于 PWM_FAULT_GROUP_1，返回值为下列值的逻辑或： ➢ PWM_FAULT_DCMP0 ➢ PWM_FAULT_DCMP1 ➢ PWM_FAULT_DCMP2 ➢ PWM_FAULT_DCMP3 ➢ PWM_FAULT_DCMP4 ➢ PWM_FAULT_DCMP5 ➢ PWM_FAULT_DCMP6 ➢ PWM_FAULT_DCMP7

表 J-16　PWMGenFaultTriggerSet

功能	配置给定 PWM 发生器故障触发的设置
函数原型	void　PWMGenFaultTriggerSet（uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Group, uint32_t ui32FaultTriggers）
参数	ui32Base　PWM 模块的基地址 ui32Gen　PWM 发生器的故障触发设置，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 ui32Group　指示被可能的故障子集的配置，这个参数必须为 PWM_FAULT_GROUP_0 或 PWM_FAULT_GROUP_1 ui32FaultTriggers　定义输入的设置有助于产生给定 PWM 发生器的故障信号 对于 PWM_FAULT_GROUP_0，返回值为下列值的逻辑或： ➢ PWM_FAULT_FAULT0 ➢ PWM_FAULT_FAULT1 ➢ PWM_FAULT_FAULT2 ➢ PWM_FAULT_FAULT3 对于 PWM_FAULT_GROUP_1，返回值为下列值的逻辑或： ➢ PWM_FAULT_DCMP0 ➢ PWM_FAULT_DCMP1 ➢ PWM_FAULT_DCMP2 ➢ PWM_FAULT_DCMP3 ➢ PWM_FAULT_DCMP4

（续）

描述	➢ PWM_FAULT_DCMP5 ➢ PWM_FAULT_DCMP6 ➢ PWM_FAULT_DCMP7
描述	对于给定的 PWM 发生器，该函数允许选择故障输入，并结合这些故障输入来产生故障状态。默认情况下，所有发生器只使用 FAULT0（向后兼容性），若用参数 ui32Config 中的 PWM_GEN_MODE_FAULT_SRC 标志来调用 PWMGenConfigure( ) 函数，将使能扩展故障处理，以及必须调用该函数来配置故障触发 在调用 PWMGenFaultConfigure( ) 配置先前设置的基础上，完成调整每个 FAULTn 输入的意义（sense）后，再对参数 ui32FaultTriggers 指定的所有信号一并进行“或运算”，即可产生 PWM 发生器的故障信号
返回参数	无

表 J-17　PWMGenIntClear

功能	清除指定 PWM 发生器模块的指定中断
函数原型	void　PWMGenIntClear（uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Ints）
参数	ui32Base　PWM 模块的基地址 ui32Gen　PWM 发生器查询，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 ui32Ints　指定待清除的中断
描述	该函数通过向中断状态寄存器中特定的位写 1 来清除指定 PWM 发生器中的指定中断 参数 ui32Ints 为下列值的逻辑或： ➢ PWM_INT_CNT_ZERO ➢ PWM_INT_CNT_LOAD ➢ PWM_INT_CNT_AU ➢ PWM_INT_CNT_AD ➢ PWM_INT_CNT_BU ➢ PWM_INT_CNT_BD
返回参数	无

表 J-18　PWMGenIntRegister

功能	注册指定 PWM 发生器模块的中断处理程序
函数原型	void PWMGenIntRegister（uint32_t ui32Base, uint32_t ui32Gen, void（ * pfnIntHandler）（void））
参数	ui32Base　PWM 模块的基地址 ui32Gen　讨论的 PWM 发生器，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 pfnIntHandler　当 PWM 发生器中断发生时指向被调函数的指针
描述	当检测到指定 PWM 发生器模块的一个中断时，该函数可确保调用由 pfnIntHandler 指定的中断处理程序。该函数也使能中断控制器中相应的 PWM 发生器中断；各个发生器中断与中断源必须调用 PWMIntEnable( ) 和 PWMGenIntTrigEnable( ) 函数来使能
返回参数	无

表 J-19　PWMGenIntStatus

功能	获取指定 PWM 发生器模块的中断状态
函数原型	uint32_t　PWMGenIntStatus（uint32_t ui32Base， uint32_t ui32Gen， bool bMasked）
参数	ui32Base　PWM 模块的基地址 ui32Gen　PWM 发生器查询，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 bMasked　指定返回屏蔽还是原始中断状态
描述	如果将 bMasked 设置为 true，则返回屏蔽中断状态；否则，将返回原始中断状态
返回参数	返回指定 PWM 发生器的中断状态寄存器的内容或原始中断状态寄存器的内容

表 J-20　PWMGenIntTrigDisable

功能	禁止指定 PWM 发生器模块的中断
函数原型	void　PWMGenIntTrigDisable（uint32_t ui32Base， uint32_t ui32Gen， uint32_t ui32IntTrig）
参数	ui32Base　PWM 模块的基地址 ui32Gen　禁止 PWM 发生器中断和触发，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 ui32IntTrig　指定待禁止的中断和触发
描述	该函数通过清除指定 PWM 发生器的中断/触发使能寄存器中的特定位来屏蔽指定的中断和触发 参数 ui32IntTrig 为下列值的逻辑或： ➢ PWM_INT_CNT_ZERO ➢ PWM_INT_CNT_LOAD ➢ PWM_INT_CNT_AU ➢ PWM_INT_CNT_AD ➢ PWM_INT_CNT_BU ➢ PWM_INT_CNT_BD ➢ PWM_TR_CNT_ZERO ➢ PWM_TR_CNT_LOAD ➢ PWM_TR_CNT_AU ➢ PWM_TR_CNT_AD ➢ PWM_TR_CNT_BU ➢ PWM_TR_CNT_BD
返回参数	无

表 J-21　PWMGenIntTrigEnable

功能	使能指定 PWM 发生器模块的中断和触发
函数原型	void PWMGenIntTrigEnable（uint32_t ui32Base， uint32_t ui32Gen， uint32_t ui32IntTrig）

（续）

参数	ui32Base　PWM 模块的基地址 ui32Gen　使能 PWM 发生器中断和触发，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 ui32IntTrig　指定待使能的中断和触发
描述	该函数通过置位指定 PWM 发生器的中断/触发使能寄存器中的特定位来取消屏蔽指定的中断和触发 参数 ui32IntTrig 为下列值的逻辑或： ➢ PWM_INT_CNT_ZERO ➢ PWM_INT_CNT_LOAD ➢ PWM_INT_CNT_AU ➢ PWM_INT_CNT_AD ➢ PWM_INT_CNT_BU ➢ PWM_INT_CNT_BD ➢ PWM_TR_CNT_ZERO ➢ PWM_TR_CNT_LOAD ➢ PWM_TR_CNT_AU ➢ PWM_TR_CNT_AD ➢ PWM_TR_CNT_BU ➢ PWM_TR_CNT_BD
返回参数	无

表 J-22　PWMGenIntUnregister

功能	移除指定 PWM 发生器模块的中断处理程序
函数原型	void PWMGenIntUnregister（uint32_t ui32Base, uint32_t ui32Gen）
参数	ui32Base　PWM 模块的基地址 ui32Gen　讨论的 PWM 发生器，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3
描述	该函数用于注销指定 PWM 发生器模块的中断处理程序。这个函数也禁止在中断控制器中相应的 PWM 发生器中断；各个发生器中断和中断源必须调用 PWMIntDisable( ) 和 PWMGenIntTrigDisable( ) 函数来禁止
返回参数	无

表 J-23　PWMGenPeriodGet

功能	获取 PWM 发生器模块的周期
函数原型	uint32_t　PWMGenPeriodGet（uint32_t ui32Base, uint32_t ui32Gen）
参数	ui32Base　PWM 模块的基地址 ui32Gen　PWM 发生器查询，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3

（续）

描述	该函数获取指定 PWM 发生器模块的周期。发生器模块的周期被定义为发生器模块中 0 信号脉冲之间的 PWM 时钟节拍数 如果指定 PWM 发生器的计数器更新还未完成，返回值可能不是现行的周期，而是用 PWM 时钟节拍测量的编程的周期
返回参数	返回 PWM 时钟节拍中指定发生器模块的可编程周期

表 J-24　PWMGenPeriodSet

功能	设置 PWM 发生器的周期
函数原型	void　PWMGenPeriodSet（uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Period）
参数	ui32Base　PWM 模块的基地址 ui32Gen　PWM 发生器查询，必须为下列值之一： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 ui32Period　指定 PWM 发生器用时钟节拍测量的输出周期
描述	该函数设置指定 PWM 发生器模块的周期，其中发生器模块的周期被定义为发生器模块中 0 信号脉冲之间的 PWM 时钟节拍数
返回参数	无

表 J-25　PWMIntDisable

功能	禁止 PWM 模块的发生器中断和故障中断
函数原型	void　PWMIntDisable（uint32_t ui32Base, uint32_t ui32GenFault）
参数	ui32Base　PWM 模块的基地址 ui32GenFault　包含要禁止的中断，参数必须为下列值的逻辑或： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 ➢ PWM_INT_FAULT0 ➢ PWM_INT_FAULT1 ➢ PWM_INT_FAULT2 ➢ PWM_INT_FAULT3
描述	该函数通过清除所选 PWM 模块的中断使能寄存器中特定的位来屏蔽指定的中断
返回参数	无

表 J-26　PWMIntEnable

功能	使能 PWM 模块的发生器中断和故障中断
函数原型	void PWMIntEnable（uint32_t ui32Base, uint32_t ui32GenFault）
参数	ui32Base　PWM 模块的基地址 ui32GenFault　包含待使能的中断，参数必须为下列值的逻辑或： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 ➢ PWM_INT_FAULT0

（续）

参数	➢ PWM_INT_FAULT1 ➢ PWM_INT_FAULT2 ➢ PWM_INT_FAULT3
描述	该函数通过置位所选 PWM 模块的中断使能寄存器中特定的位来取消屏蔽指定的中断
返回参数	无

表 J-27　PWMIntStatus

功能	获取指定 PWM 模块的中断状态
函数原型	uint32_t PWMIntStatus（uint32_t ui32Base, bool bMasked）
参数	ui32Base　PWM 模块的基地址 bMasked　指定返回屏蔽还是原始中断状态
描述	如果将 bMasked 设置为 true，则返回屏蔽中断状态；否则，将返回原始中断状态
返回参数	返回当前的中断状态，且枚举为下列的位字段形式： ➢ PWM_GEN_0 ➢ PWM_GEN_1 ➢ PWM_GEN_2 ➢ PWM_GEN_3 ➢ PWM_INT_FAULT0 ➢ PWM_INT_FAULT1 ➢ PWM_INT_FAULT2 ➢ PWM_INT_FAULT3

表 J-28　PWMOutputFault

功能	指定 PWM 输出的状态来响应故障状态
函数原型	void　PWMOutputFault（uint32_t ui32Base, uint32_t ui32PWMOutBits, bool bFaultSuppress）
参数	ui32Base　PWM 模块的基地址 ui32PWMOutBits　待修改的 PWM 输出，为下列值的逻辑或： ➢ PWM_OUT_0_BIT ➢ PWM_OUT_1_BIT ➢ PWM_OUT_2_BIT ➢ PWM_OUT_3_BIT ➢ PWM_OUT_4_BIT ➢ PWM_OUT_5_BIT ➢ PWM_OUT_6_BIT ➢ PWM_OUT_7_BIT bFaultSuppress　确定在一个激活的故障状态期间信号是被抑制还是通过
描述	该函数用于设置所选 PWM 输出的故障处理特性。使用参数 ui32PWMOutBits 来选择输出。参数 bFaultSuppress 决定所选输出的故障处理特性。如果将 bFaultSuppress 设置为 true，则所选输出无效。若 bfaultsuppress 为 false，则选定的输出不受检测到故障的影响 在支持扩展 PWM 故障处理的设备上，受影响的输出引脚的驱动状态，可由 PWMOutput-FaultLevel( ) 函数来配置。如果不配置，或者如果设备不支持扩展 PWM 故障处理，则在故障状态中受影响的输出引脚将被驱动为低电平
返回参数	无

表 J-29　PWMOutputFaultLevel

功能	指定被抑制的 PWM 输出电平来响应一个故障状态
函数原型	void　PWMOutputFaultLevel（uint32_t ui32Base, uint32_t ui32PWMOutBits, bool bDriveHigh）
参数	ui32Base　PWM 模块的基地址 ui32PWMOutBits　待修改的 PWM 输出，为下列值的逻辑或： ➢ PWM_OUT_0_BIT ➢ PWM_OUT_1_BIT ➢ PWM_OUT_2_BIT ➢ PWM_OUT_3_BIT ➢ PWM_OUT_4_BIT ➢ PWM_OUT_5_BIT ➢ PWM_OUT_6_BIT ➢ PWM_OUT_7_BIT bDriveHigh　确定在一个激活的故障状态期间信号是被拉高还是拉低
描述	该函数决定被抑制的 PWM 输出引脚在响应故障状态时是被驱动为高电平还是低电平。使用参数 ui32PWMOutBits 来选择受影响的输出。参数 bDriveHigh 决定由 ui32PWMOutBits 辨识的引脚输出电平。如果将 bDriveHigh 设置为 true，则当检测到一个故障时所选的输出将被驱动为高电平。若 bDriveHigh 为 false，则引脚被驱动为低电平 在故障状态下，通过调用 PWMOutputFault( )未将其配置成抑制的引脚，将不受这个函数影响
返回参数	无

表 J-30　PWMOutputInvert

功能	选择 PWM 输出的反相模式
函数原型	void　PWMOutputInvert（uint32_t ui32Base, uint32_t ui32PWMOutBits, bool bInvert）
参数	ui32Base　PWM 模块的基地址 ui32PWMOutBits　待修改的 PWM 输出，为下列值的逻辑或： ➢ PWM_OUT_0_BIT ➢ PWM_OUT_1_BIT ➢ PWM_OUT_2_BIT ➢ PWM_OUT_3_BIT ➢ PWM_OUT_4_BIT ➢ PWM_OUT_5_BIT ➢ PWM_OUT_6_BIT ➢ PWM_OUT_7_BIT bInvert　确定信号是被反相还是通过
描述	该函数用于为选定的 PWM 输出选择反相模式。使用参数 ui32PWMOutBits 选择输出，而参数 bInvert 用于确定所选输出的反相模式。如果将 bInvert 设置为 true，则指定的 PWM 输出信号将反相或使低电平有效。若 bInvert 为 false，则指定的输出信号将通过或使高电平有效
返回参数	无

表 J-31　PWMOutputState

功能	使能或禁止 PWM 输出
函数原型	void　PWMOutputState（uint32_t　ui32Base, uint32_t ui32PWMOutBits, bool bEnable）

（续）

参数	ui32Base　PWM 模块的基地址 ui32PWMOutBits　待修改的 PWM 输出，为下列值的逻辑或： ➢ PWM_OUT_0_BIT ➢ PWM_OUT_1_BIT ➢ PWM_OUT_2_BIT ➢ PWM_OUT_3_BIT ➢ PWM_OUT_4_BIT ➢ PWM_OUT_5_BIT ➢ PWM_OUT_6_BIT ➢ PWM_OUT_7_BIT bEnable　确定信号是使能还是禁止
描述	该函数使能或禁止所选 PWM 输出。使用参数 ui32PWMOutBits 选择输出。而参数 bEnable 用于确定所选输出的状态。如果将 bEnable 设置为 true，则使能所选的 PWM 输出，或将其置于激活状态。如果 bEnable 为 false，则选定的输出将被禁止，或将其置于非激活状态
返回参数	无

表 J-32　PWMOutputUpdateMode

功能	设置 PWM 输出的更新模式或同步模式
函数原型	void　PWMOutputUpdateMode（uint32_t ui32Base, uint32_t ui32PWMOutBits, uint32_t ui32Mode）
参数	ui32Base　PWM 模块的基地址 ui32PWMOutBits　待修改的 PWM 输出，为下列值的逻辑或： ➢ PWM_OUT_0_BIT ➢ PWM_OUT_1_BIT ➢ PWM_OUT_2_BIT ➢ PWM_OUT_3_BIT ➢ PWM_OUT_4_BIT ➢ PWM_OUT_5_BIT ➢ PWM_OUT_6_BIT ➢ PWM_OUT_7_BIT ui32Mode　当使能或禁止 PWM 输出时，指定使能的更新使用模式
描述	该函数用于设置三个候选更新模式中的一种来使能或禁止 PWM 输出请求。参数 ui32Mode 控制所做的更改需通过调用 PWMOutputState（）来生效。其可能的值如下： ➢ PWM_OUTPUT_MODE_NO_SYNC　　//使能/禁止的变更立即生效 ➢ PWM_OUTPUT_MODE_SYNC_LOCAL　　//当本地 PWM 发生器中的计数器在下一次到达 0 时，使改变生效 ➢ PWM_OUTPUT_MODE_SYNC_GLOBAL　　//当本地 PWM 发生器中的计数器在下一次到达 0，随后调用 PWMSyncUpdate（）在其 ui32GenBits 参数中指定相同的发生器，使改变生效
返回参数	无

表 J-33　PWMPulseWidthGet

功能	获取 PWM 输出的脉冲宽度
函数原型	uint32_t　PWMPulseWidthGet（uint32_t　ui32Base, uint32_t ui32PWMOut）
参数	ui32Base　PWM 模块的基地址 ui32PWMOut　PWM 输出查询，参数为下列值之一： ➢ PWM_OUT_0

（续）

参数	➢ PWM_OUT_1 ➢ PWM_OUT_2 ➢ PWM_OUT_3 ➢ PWM_OUT_4 ➢ PWM_OUT_5 ➢ PWM_OUT_6 ➢ PWM_OUT_7
描述	该函数用于获取指定 PWM 输出的当前可编程脉冲宽度。如果指定 PWM 发生器的计数器更新还未完成，返回值可能不是现行的周期，而是用 PWM 时钟节拍测量的编程的周期
返回参数	返回 PWM 时钟节拍的脉冲宽度

表 J-34　PWMPulseWidthSet

功能	设置指定 PWM 输出的脉冲宽度
函数原型	void　PWMPulseWidthSet（uint32_t ui32Base, uint32_t ui32PWMOut, uint32_t ui32Width）
参数	ui32Base　PWM 模块的基地址 ui32PWMOut　PWM 输出查询，参数为下列值之一： ➢ PWM_OUT_0 ➢ PWM_OUT_1 ➢ PWM_OUT_2 ➢ PWM_OUT_3 ➢ PWM_OUT_4 ➢ PWM_OUT_5 ➢ PWM_OUT_6 ➢ PWM_OUT_7 ui32Width　指定脉冲正部分的宽度
描述	该函数设置指定 PWM 输出的脉冲宽度，其中脉冲宽度被定义为 PWM 时钟节拍数
返回参数	无

表 J-35　PWMSyncTimeBase

功能	让一个或多个 PWM 发生器模块的计数器同步计数
函数原型	void　PWMSyncTimeBase（uint32_t ui32Base, uint32_t ui32GenBits）
参数	ui32Base　PWM 模块的基地址 ui32GenBits　待同步的 PWM 发生器模块，为下列值的逻辑或： ➢ PWM_GEN_0_BIT ➢ PWM_GEN_1_BIT ➢ PWM_GEN_2_BIT ➢ PWM_GEN_3_BIT
描述	对于所选定的 PWM 模块，该函数通过让指定发生器中的计数器复位来同步发生器模块的时基
返回参数	无

表 J-36　PWMSyncUpdate

功能	同步所有等待的更新
函数原型	void　PWMSyncUpdate（uint32_t ui32Base, uint32_t ui32GenBits）

（续）

参数	ui32Base　PWM 模块的基地址 ui32GenBits　待更新的 PWM 发生器模块，为下列值的逻辑或： ➢ PWM_GEN_0_BIT ➢ PWM_GEN_1_BIT ➢ PWM_GEN_2_BIT ➢ PWM_GEN_3_BIT
描述	对于选定的 PWM 发生器，该函数使所有排队的周期或脉冲宽度的更新在下一次相应计数器变为 0 时进行
返回参数	无



附录 K　第 13 章附录：μDMA 固件库函数简介

本章附录将介绍 μDMA 的固件库函数，包括：定义文档和函数文档两个部分。

1. 定义文档

介绍 μDMA 固件库的定义文档。

表 K-1　μDMATaskStructEntry

功能	构建散聚任务表项的辅助宏
定义	#define μDMATaskStructEntry（ui32TransferCount, ui32ItemSize, ui32SrcIncrement, pvSrcAddr, ui32DstIncrement, pvDstAddr, ui32ArbSize, ui32Mode）
参数	ui32TransferCount　传输任务的项目计数 ui32ItemSize　传输任务的项目位大小 ui32SrcIncrement　源数据位大小的增量 pvSrcAddr　传输数据的起始地址 ui32DstIncrement　目标数据位大小的增量 pvDstAddr　目标数据的起始地址 ui32ArbSize　传输任务的仲裁大小 ui32Mode　任务的传输模式
描述	这个宏的目的是用来帮助构建散聚传输的一个 μDMA 任务表。这个宏将根据输入参数计算任务结构体成员（entry）字段的值 对每个参数的值有特殊要求。由于不做检查，所以调用者要保证使用的参数值是正确的 ➢ 参数 ui32TransferCount 待传输的任务项目数量。它的值必须在 1～1024 之间 ➢ 参数 ui32ItemSize 待传输数据的位大小。它必须是 UDMA_SIZE_8、UDMA_SIZE_16，或者 UDMA_SIZE_32 中的一个值 ➢ 参数 ui32SrcIncrement　//是源数据增加的大小。它的值必须是 UDMA_SRC_INC_8、UDMA_SRC_INC_16、UDMA_SRC_INC_32，或者 UDMA_SRC_INC_NONE 中的一个值 ➢ 参数 pvSrcAddr　指向源数据起始地址的空指针 ➢ 参数 ui32DstIncrement　目标数据的增量大小。它必须是 UDMA_DST_INC_8、UDMA_DST_INC_16、UDMA_DST_INC_32，或者 UDMA_DST_INC_NONE 中的一个值 ➢ 参数 pvDstAddr　指向将要传输的目标数据的初始地址的空指针 ➢ 参数 ui32ArbSize 传输数据的仲裁大小，它的值必须是 UDMA_ARB_1、UDMA_ARB_2、UDMA_ARB_4 … UDMA_ARB_1024 中的一个。此参数为 2 的幂用于选择仲裁大小，其值为 1～1024

（续）

描述	➢ 参数 ui32Mode 传输任务的模式。它必须是 UDMA_MODE_BASIC、UDMA_MODE_AUTO、UDMA_MODE_MEM_SCATTER_GATHER，或者 UDMA_MODE_PER_SCATTER_GATHER 中的一个值 例如，宏可用来初始化 tDMAControlTable 结构体中的各个成员： tDMAControlTable MyTaskList[ ] = { uDMATaskStructEntry(Task1Count, UDMA_SIZE_8 UDMA_SRC_INC_8, MySourceBuf, UDMA_DST_INC_8, MyDestBuf, UDMA_ARB_8, UDMA_MODE_MEM_SCATTER_GATHER), uDMATaskStructEntry （Task2Count, ...), }
返回参数	无

2. 函数文档

介绍 μDMA 的固件库库函数的功能及定义。

表 K-2 μDMAChannelAssign

功能	分配 μDMA 通道的外设映射
函数原型	void μDMAChannelAssign（uint32_t ui32Mapping）
参数	ui32Mapping 指定通道外设分配的宏
描述	该函数给 μDMA 通道分配一个外设映射，用于选择哪个外设为 μDMA 通道。参数 ui32Mapping 为头文件 udma. h 中命名为 UDMA_CHn_tttt 中的一个宏。例如，把 μDMA 通道 0 分配给 UART2 RX 通道，则参数就应该为宏 UDMA_CH0_UART2RX
返回参数	无

表 K-3 μDMAChannelAttributeDisable

功能	禁止 μDMA 通道的属性
函数原型	void μDMAChannelAttributeDisable（uint32_t ui32ChannelNum, uint32_t ui32Attr）
参数	ui32ChannelNum 待配置的通道 ui32Attr 通道的属性组合
描述	该函数用于禁止 μDMA 通道的属性 参数 ui32Attr 为以下任意值的逻辑或： ➢ UDMA_ATTR_USEBURST //用于限制传输仅适用突发模式 ➢ UDMA_ATTR _ALTSELECT //用于选择该通道的副控制结构 ➢ UDMA_ATTR_HIGH_PRIORITY //用于设置该信道为高优先级 ➢ UDMA_ATTR_REQMASK //屏蔽该通道来自外设的硬件请求信号
返回参数	无

表 K-4　μDMAChannelAttributeEnable

功能	使能 μDMA 通道的属性
函数原型	void　μDMAChannelAttributeEnable（uint32_t ui32ChannelNum, uint32_t ui32Attr）
参数	ui32ChannelNum　待配置的通道 ui32Attr　通道的属性组合
描述	该函数用于使能一个 μDMA 通道的属性 参数 ui32Attr 为以下任意值的逻辑或： ➢ UDMA_ATTR_USEBURST　　//用于限制传输仅适用突发模式 ➢ UDMA_ATTR _ALTSELECT　　//用于选择该通道的副控制结构 ➢ UDMA_ATTR_HIGH_PRIORITY //用于设置该信道为高优先级 ➢ UDMA_ATTR_REQMASK　　//屏蔽该通道来自外设的硬件请求信号
返回参数	无

表 K-5　μDMAChannelAttributeGet

功能	获取 μDMA 通道的属性使能
函数原型	μint32_t μDMAChannelAttributeGet（uint32_t　ui32ChannelNum）
参数	ui32ChannelNum　待配置的通道
描述	该函数返回代表 μDMA 通道属性标志的组合
返回参数	返回 μDMA 通道属性的逻辑或，为下列值之一： ➢ UDMA_ATTR_USEBURST　　//用于限制传输仅适用突发模式 ➢ UDMA_ATTR _ALTSELECT　　//用于选择该通道的副控制结构 ➢ UDMA_ATTR_HIGH_PRIORITY　//用于设置该信道为高优先级 ➢ UDMA_ATTR_REQMASK　　//屏蔽该通道来自外设的硬件请求信号

表 K-6　μDMAChannelControlSet

功能	设置 μDMA 通道控制结构的控制参数
函数原型	void μDMAChannelControlSet（uint32_t ui32ChannelStructIndex, uint32_t ui32Control）
参数	ui32ChannelStructIndex　μDMA 通道号与 UDMA_PRI_SELECT 或 UDMA_ALT_SELECT 的逻辑或 ui32Control　设置通道控制参数的几个控制值的逻辑或
描述	该函数用于设置 μDMA 传输的控制参数。这些参数是典型值，通常不会改变 ➢ 参数 ui32ChannelStructIndex 应该是信道号与 UDMA_PRI_SELECT 或 UDMA_ALT_SELECT 之一的逻辑或，以选择是使用主数据结构还是副数据结构 ➢ 参数 ui32Control 是以下五个值的逻辑或：数据大小、源地址增量、目标地址增量、仲裁大小以及使用突发标志 每个这些值的可用选择描述如下： ■ 从 UDMA_SIZE_8、UDMA_SIZE_16 或 UDMA_SIZE_32 中选择一个数据大小，来选取大小为 8、16 或者 32 位的数据 ■ 从 UDMA_SRC_INC_8、UDMA_SRC_INC_16、UDMA_SRC_INC_32 或者 UDMA_SRC_INC_NONE 中选择一个源地址增量大小，来选取 8 位字节、16 位半字、32 位字或者选择无地址增量 ■ 从 UDMA_DST_INC_8、UDMA_DST_INC_16、UDMA_DST_INC_32 或者 UDMA_DST_INC_NONE 中选择一个目标地址增量，可选取大小为 8 位字节、16 位半个字、32 位字或者选择无地址增量 ■ 仲裁大小决定了在 μDMA 控制器重新仲裁总线之前传输了多少个项目。可从 UDMA_ARB_1、UDMA_ARB_2、UDMA_ARB_4、UDMA_ARB_8、…UDMA_ARB_1024 的 1 ~ 1024 个项目中选择仲裁的大小，其值为 2 的幂 ■ 值 UDMA_NEXT_USEBURST 用于强制通道在散聚传输的末端只响应突发请求
返回参数	无

表 K-7　μDMAChannelDisable

功能	禁止 μDMA 通道的操作
函数原型	void μDMAChannelDisable（uint32_t　ui32ChannelNum）
参数	ui32ChannelNum　待禁止的信道号
描述	该函数禁止特定的 μDMA 通道。一旦禁止，该通道只有通过 μDMAChannelEnable( ) 函数对其重新使能后，方可对 μDMA 传输请求做出响应
返回参数	无

表 K-8　μDMAChannelEnable

功能	使能 μDMA 通道的操作
函数原型	void　μDMAChannelEnable（uint32_t　ui32ChannelNum）
参数	ui32ChannelNum　待使能的信道号。
描述	该函数使能一个使用的特定的 μDMA 通道。并且该函数必须在通道执行 μDMA 传输之前使能 当一个 μDMA 传输完成时，μDMA 控制器将自动禁止通该道。因此，这个函数应该在启动任何新的传输前先行调用
返回参数	无

表 K-9　μDMAChannelIsEnabled

功能	检查是否使能一个 μDMA 通道的操作
函数原型	bool μDMAChannelIsEnabled（uint32_t　ui32ChannelNum）
参数	ui32ChannelNum　待检查的信道号
描述	该函数用于检查一个特定的 μDMA 信道是否被使能。在一个传输结束自动禁止该通道传输时，这个函数可用来检查传输的状态
返回参数	如果返回 true，则通道使能；如果返回 false，则禁止通道传输

表 K-10　μDMAChannelModeGet

功能	获取一个 μDMA 通道控制结构的传输模式
函数原型	uint32_t uDMAChannelModeGet（uint32_t　ui32ChannelStructIndex）
参数	ui32ChannelStructIndex　μDMA 通道号与 UDMA_PRI_SELECT 或 UDMA_ALT_SELECT 之一的逻辑或
描述	该函数用于获取 μDMA 通道的传输模式，并且查询通道上的传输状态。当传输完成时模式将为 UDMA_MODE_STOP
返回参数	返回指定通道传输模式和控制结构，为下列值之一： ➢ UDMA_MODE_STOP ➢ UDMA_MODE_BASIC ➢ UDMA_MODE_AUTO ➢ UDMA_MODE_PINGPONG ➢ UDMA_MODE_MEM_SCATTER_GATHER ➢ UDMA_MODE_PER_SCATTER_GATHER

表 K-11　μDMAChannelRequest

功能	请求 μDMA 通道启动传输
函数原型	void μDMAChannelRequest（uint32_t　ui32ChannelNum）
参数	ui32ChannelNum　请求 μDMA 传输的通道号
描述	该函数允许软件请求一个 μDMA 通道开始传输。该函数可用于执行存储器→存储器的传输，或由于某些原因，一次传输需由软件启动，而非与该通道相关联的外设来开始
返回参数	无

表 K-12　μDMAChannelScatterGatherSet

功能	配置 μDMA 通道的散聚模式
函数原型	void μDMAChannelScatterGatherSet（uint32_t ui32ChannelNum, uint32_t ui32TaskCount, void ＊pvTaskList, uint32_t ui32IsPeriphSG）
参数	ui32ChannelNum　μDMA 通道号 ui32TaskCount　待执行的散聚任务数 pvTaskList　指向散聚任务列表开始的指针 ui32IsPeriphSG　指示一个外设散聚传输的标志（否则为存储器散聚传输）
描述	该函数用于配置散聚模式的通道。调用者必须先行建立一个任务列表，且必须传递参数 pvTaskList 指向任务列表的开始。参数 ui32TaskCount 为任务列表中的任务计数，而非任务列表的大小。标志 bIsPeriphSG 用于指示散聚是配置为外设操作还是存储器操作
返回参数	无

表 K-13　μDMAChannelSelectDefault

功能	选择默认外设的一组 μDMA 通道
函数原型	void μDMAChannelSelectDefault（uint32_t ui32DefPeriphs）
参数	ui32DefPeriphs　使用默认外设的 μDMA 通道的逻辑或，而非第二外设
描述	该函数用于选择默认外设分配的一组 μDMA 通道 参数 ui32DefPeriphs 为下列任意宏的逻辑或： ➢ UDMA_DEF_USBEP1RX_SEC_UART2RX ➢ UDMA_DEF_USBEP1TX_SEC_UART2TX ➢ UDMA_DEF_USBEP2RX_SEC_TMR3A ➢ UDMA_DEF_USBEP2TX_SEC_TMR3B ➢ UDMA_DEF_USBEP3RX_SEC_TMR2A ➢ UDMA_DEF_USBEP3TX_SEC_TMR2B ➢ UDMA_DEF_ETH0RX_SEC_TMR2A ➢ UDMA_DEF_ETH0TX_SEC_TMR2B ➢ UDMA_DEF_UART0RX_SEC_UART1RX ➢ UDMA_DEF_UART0TX_SEC_UART1TX ➢ UDMA_DEF_SSI0RX_SEC_SSI1RX ➢ UDMA_DEF_SSI0TX_SEC_SSI1TX ➢ UDMA_DEF_RESERVED_SEC_UART2RX ➢ UDMA_DEF_RESERVED_SEC_UART2TX ➢ UDMA_DEF_ADC00_SEC_TMR2A ➢ UDMA_DEF_ADC01_SEC_TMR2B ➢ UDMA_DEF_ADC02_SEC_RESERVED ➢ UDMA_DEF_ADC03_SEC_RESERVED ➢ UDMA_DEF_TMR0A_SEC_TMR1A ➢ UDMA_DEF_TMR0B_SEC_TMR1B ➢ UDMA_DEF_TMR1A_SEC_EPI0RX ➢ UDMA_DEF_TMR1B_SEC_EPI0TX ➢ UDMA_DEF_UART1RX_SEC_RESERVED

（续）

描述	➢ UDMA_DEF_UART1TX_SEC_RESERVED ➢ UDMA_DEF_SSI1RX_SEC_ADC10 ➢ UDMA_DEF_SSI1TX_SEC_ADC11 ➢ UDMA_DEF_RESERVED_SEC_ADC12 ➢ UDMA_DEF_RESERVED_SEC_ADC13 ➢ UDMA_DEF_I2S0RX_SEC_RESERVED ➢ UDMA_DEF_I2S0TX_SEC_RESERVED
返回参数	无

表 K-14　μDMAChannelSelectSecondary

功能	选择一组 μDMA 通道的第二外设
函数原型	void μDMAChannelSelectSecondary（uint32_t　ui32SecPeriphs）
参数	ui32SecPeriphs　使用第二外设的 μDMA 通道的逻辑或，而非默认的外设
描述	该函数用于为一组 μDMA 通道选择第二外设分配。通过选择通道的第二外设分配，使默认的外设不再对该通道有效 参数 ui32SecPeriphs 为下列任意宏的逻辑或： ➢ UDMA_DEF_USBEP1RX_SEC_UART2RX ➢ UDMA_DEF_USBEP1TX_SEC_UART2TX ➢ UDMA_DEF_USBEP2RX_SEC_TMR3A ➢ UDMA_DEF_USBEP2TX_SEC_TMR3B ➢ UDMA_DEF_USBEP3RX_SEC_TMR2A ➢ UDMA_DEF_USBEP3TX_SEC_TMR2B ➢ UDMA_DEF_ETH0RX_SEC_TMR2A ➢ UDMA_DEF_ETH0TX_SEC_TMR2B ➢ UDMA_DEF_UART0RX_SEC_UART1RX ➢ UDMA_DEF_UART0TX_SEC_UART1TX ➢ UDMA_DEF_SSI0RX_SEC_SSI1RX ➢ UDMA_DEF_SSI0TX_SEC_SSI1TX ➢ UDMA_DEF_RESERVED_SEC_UART2RX ➢ UDMA_DEF_RESERVED_SEC_UART2TX ➢ UDMA_DEF_ADC00_SEC_TMR2A ➢ UDMA_DEF_ADC01_SEC_TMR2B ➢ UDMA_DEF_ADC02_SEC_RESERVED ➢ UDMA_DEF_ADC03_SEC_RESERVED ➢ UDMA_DEF_TMR0A_SEC_TMR1A ➢ UDMA_DEF_TMR0B_SEC_TMR1B ➢ UDMA_DEF_TMR1A_SEC_EPI0RX ➢ UDMA_DEF_TMR1B_SEC_EPI0TX ➢ UDMA_DEF_UART1RX_SEC_RESERVED ➢ UDMA_DEF_UART1TX_SEC_RESERVED ➢ UDMA_DEF_SSI1RX_SEC_ADC10 ➢ UDMA_DEF_SSI1TX_SEC_ADC11 ➢ UDMA_DEF_RESERVED_SEC_ADC12 ➢ UDMA_DEF_RESERVED_SEC_ADC13 ➢ UDMA_DEF_I2S0RX_SEC_RESERVED ➢ UDMA_DEF_I2S0TX_SEC_RESERVED
返回参数	无

表 K-15　μDMAChannelSizeGet

功能	获取 μDMA 通道控制结构的当前传输大小
函数原型	uint32_t μDMAChannelSizeGet（uint32_t ui32ChannelStructIndex）

（续）

参数	ui32ChannelStructIndex　μDMA 信道号与 UDMA_PRI_SELECT 或 UDMA_ALT_SELECT 之一的逻辑或
描述	该函数用于获取通道的 μDMA 传输大小。传输的大小就是待传输的项目个数，其中项目大小可能是 8、16 或 32 位。如果只发生了部分传输，则将返回剩余项目的个数。如果传输完成，则返回值为 0
返回参数	返回待传输剩余项目的个数

表 K-16　μDMAChannelTransferSet

功能	设置 μDMA 通道控制结构的传输参数
函数原型	void μDMAChannelTransferSet（uint32_t ui32ChannelStructIndex, uint32_t ui32Mode, void ＊pvSrcAddr, void ＊pvDstAddr, uint32_t ui32TransferSize）
参数	ui32ChannelStructIndex　μDMA 信道号与 UDMA_PRI_SELECT 或者 UDMA_ALT_SELECT 的逻辑或 ui32Mode　μDMA 传输的类型 pvSrcAddr　传输的源地址 pvDstAddr　传输的目标地址 ui32TransferSize　待传输的数据项目个数
描述	该函数用于配置 μDMA 传输的参数。通常这些参数是经常改变的。该通道在调用这个函数之前，必须至少先行调用 μDMAChannelControlSet( ) 一次 参数 ui32ChannelStructIndex 应为通道号与 UDMA_PRI_SELECT 或 UDMA_ALT_SELECT 的逻辑或，以选择是主数据结构还是副数据结构 ➢ 参数 ui32Mode 应为下列值之一： ■ UDMA_MODE_STOP　//停止 μDMA 传输。在传输结 束时控制器将模式设定为该值 ■ UDMA_MODE_BASIC //执行基于请求的基本传输 ■ UDMA_MODE_AUTO//传输一旦开始，执行的传输总是会完成，即使取消请求 ■ UDMA_MODE_PINGPONG //建立一个在主控制结构 与副控制结构之间切换的传输通道，该模式允许在 uDMA 传输中使用乒乓缓冲 ■ UDMA_MODE_MEM_SCATTER_GATHER //建立一个存储器散聚传输 ■ UDMA_MODE_PER_SCATTER_GATHER//建立一个外设散聚传输 ➢ 参数 pvSrcAddr 和 pvDstAddr 指向待传输数据存储单元首地址的指针。这些地址应根据项目大小对齐。如果指针是指向适当数据类型的存储单元，编译器将负责这种对齐操作 ➢ 参数 ui32TransferSize 数据的项目个数，而非字节数 两种散聚模式，即存储器和外设实际上是不同的，这取决于选择主控制结构还是副控制结构。该函数寻找 UDMA_PRI_SELECT 和 UDMA_ALT_SELECT 的标志与通道号，并为主控制结构和副控制结构设置适当的散聚模式 在调用该函数之后，通道必须调用 μDMAChannelEnable( ) 来使能。只有在配置好通道与通道使能之后，才能进行数据传输。注意，通道在传输完成之后将自动关闭。即在建立好下次传输之后，必须再调用一次 μDMAChannelEnable( ) 函数来使能传输通道
返回参数	无

表 K-17　μDMAControlAlternateBaseGet

功能	获取通道控制表副结构的基地址
函数原型	void ＊μDMAControlAlternateBaseGet（void）
描述	该函数获取用于保存副控制结构的通道控制表下半部分的基地址
返回参数	返回一个指向通道控制表下半部分的基地址

表 K-18　uDMAControlBaseGet

功能	获取通道控制的表基地址
函数原型	void　* uDMAControlBaseGet（void）
描述	该函数获取通道控制表的基地址。这个表驻留在系统存储器中，并且用于保存每个 μDMA 通道的控制信息
返回参数	返回指向通道控制表的基地址的指针

表 K-19　uDMAControlBaseSet

功能	设置通道控制表的基地址
函数原型	void uDMAControlBaseSet（void　* psControlTable）
参数	psControlTable　指向 μDMA 通道控制表中以 1024 字节对齐的基地址的指针
描述	该函数配置通道控制表的基地址。该表驻留在系统存储器中，用于保存每个 μDMA 通道的控制信息。此表必须以 1024 字节的边界对齐，并且基地址必须在使用任何通道函数之前配置 通道控制表的大小由 μDMA 通道的个数和使用的传输模式决定
返回参数	无

表 K-20　uDMADisable

功能	禁止 μDMA 控制器的使用
函数原型	void　uDMADisable（void）
描述	该函数禁止 μDMA 控制器。一旦被禁止，在未调用 uDMAEnable( )来重新使能 μDMA 控制器之前，该 μDMA 控制器不能被操作
返回参数	无

表 K-21　uDMAEnable

功能	使能 μDMA 控制器的使用
函数原型	void uDMAEnable（void）
描述	该函数使能 μDMA 控制器。在配置和使用 μDMA 之前，必须首先使能 μDMA 控制器
返回参数	无

表 K-22　uDMAErrorStatusClear

功能	清除 μDMA 的错误中断
函数原型	void uDMAErrorStatusClear（void）
描述	该函数清除挂起的 μDMA 错误中断。应在 μDMA 错误中断处理程序中调用该函数来清除中断
返回参数	无

表 K-23　uDMAErrorStatusGet

功能	获取 μDMA 的错误状态
函数原型	uint32_t uDMAErrorStatusGet（void）
描述	该函数返回 μDMA 的错误状态。应在 μDMA 错误中断处理程序中调用该函数，以确定是否发生了 μDMA 错误
返回参数	如果 μDMA 错误挂起，则返回非零值

表 K-24　uDMAIntClear

功能	清除 μDMA 的中断状态
函数原型	void uDMAIntClear（uint32_t ui32ChanMask）
参数	ui32ChanMask　32 位屏蔽位，每一个位对应一个 μDMA 通道
描述	该函数清除在 ui32ChanMask 设置的 μDMA 中断状态寄存器中的位。每个通道对应其中的一位。如果一个位在 ui32ChanMask 中置位，则相应通道的中断状态将被清除（如果它被置位）
返回参数	无

表 K-25　uDMAIntRegister

功能	注册 μDMA 控制器的中断处理程序
函数原型	void uDMAIntRegister（uint32_t　ui32IntChannel, void（ * pfnHandler）（void））
参数	ui32IntChannel　确定待注册的 μDMA 中断 pfnHandler　指向在中断激活时被调函数的指针
描述	该函数在 μDMA 控制器产生中断时注册和使能被调用的处理程序 参数 ui32IntChannel 为下值之一： ➢ UDMA_INT_SW　//注册中断处理程序来处理 μDMA 软件通道的中断（UDMA_CHANNEL_SW） ➢ UDMA_INT_SW　//注册中断处理程序来处理 μDMA 的错误中断
返回参数	无

表 K-26　uDMAIntStatus

功能	获取 μDMA 控制器通道中断状态
函数原型	uint32_t uDMAIntStatus（void）
描述	该函数用于获取 μDMA 控制器的中断状态。返回一个 32 位的位屏蔽，它表明哪个通道正在请求中断。该函数可以用来在中断处理程序中决定或确认哪个 μDMA 信道请求了一个中断
返回参数	返回一个 32 位屏蔽，以指示请求的 μDMA 通道。每个通道对应有一个位，1 表示该信道有中断请求，且可以置位多个位

表 K-27　uDMAIntUnregister

功能	注销 μDMA 控制器的中断处理程序
函数原型	void uDMAIntUnregister（uint32_t ui32IntChannel）
参数	ui32IntChannel　确定待注销的 μDMA 中断
描述	该函数用于禁止和注销指定 μDMA 中断调用的中断处理程序。参数 ui32IntChannel 为 UDMA_INT_SW 或 UDMA_INT_ERR 其中之一，见表 K-25 中的介绍
返回参数	无



附录 L　第 14 章附录：USB DriverLib 固件库函数简介

本章附录将介绍通用 USB API 函数的用法，详细的 USB 固件库请参考 TI 的 TivaWare™ USB Library USER'S GUIDE 文档（见文档：SW－TM4C－USBL－UG－2.0.1.11577）。

表 L-1　USBClockDisable

功能	禁止 USB 控制器 PHY 的时钟
函数原型	void　USBClockDisable（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块基地址
描述	该函数禁止 USB PHY 的时钟输入或输出
返回	无

表 L-2　USBClockEnable

功能	配置和使能 USB 控制器 PHY 的时钟
函数原型	void　USBClockEnable（uint32_t ui32Base, uint32_t ui32Div, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块基地址 ui32Div　指定输入时钟的分频比 ui32Flags　指定 USB 时钟使用哪个时钟
描述	函数配置和使能 USB PHY 时钟作为 USB 控制器的输入，或者作为与输出外部连接到 USB PHY。ui32Flags 参数可用下列值指定时钟源： ➢ USB_CLOCK_INTERNAL　　//使用内部时钟作为 USB PHY 的时钟源 ➢ USB_CLOCK_EXTERNAL　　//指定 USB0CLK 输入引脚作为 USB PHY 的时钟源 如果指定 USB_CLOCK_INTERNAL，则参数 ui32Div 用来指定内部时钟的分频比；如果指定 USB_CLOCK_EXTERNAL，那么将忽略 ui32Div。当指定 USB_CLOCK_INTERNAL 时，则必须设置 ui32Div 的值，以便由 PLL_VCO/ui32Div 得到 60 MHz 时钟
返回	当前的设备地址

表 L-3　USBControllerVersion

功能	返回该 USB 控制器的版本
函数原型	uint32_t　USBControllerVersion（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块基地址
描述	该函数返回 USB 控制器的版本号，可用于调整 Tiva 系列中 USB 控制器之间的细微差别。所返回的值为 USB_CONTROLLER_VER_0 及 USB_CONTROLLER_VER_1
返回	这个函数返回 USB_CONTROLLER_VER_ values 中的值之一

表 L-4　USBDevAddrGet

功能	返回在设备模式中的当前设备地址
函数原型	uint32_t USBDevAddrGet（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块基地址
描述	该函数返回当前设备的地址。该地址可通过调用 USBDevAddrSet( )来设置
返回	当前的设备地址

表 L-5　USBDevAddrSet

功能	设置在设备模式下的地址
函数原型	void　USBDevAddrSet（uint32_t ui32Base，uint32_t ui32Address）
参数	ui32Base　指定 USB 模块基地址 ui32Address　设备待使用的地址
描述	该函数用于配置 USB 总线上的设备地址。这个地址很可能是接收来自主机控制器的地址设置命令
返回	无

表 L-6　USBDevConnect

功能	在设备模式中把 USB 控制器连接到总线上
函数原型	void　USBDevConnect（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块基地址
描述	该函数使能 USB 控制器的软连接功能。调用 USBDevDisconnect( )就可从总线上拿掉 USB 设备
返回	无

表 L-7　USBDevDisconnect

功能	在设备模式中从总线上移除 USB 控制器
函数原型	void　USBDevDisconnect（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块基地址
描述	该函数使用 USB 控制器的软连接功能来清除 USB 总线上的设备。调用 USBDevConnect( )可重新使设备连接到总线上
返回	无

表 L-8　USBDevEndpointConfigGet

功能	获取一个端点的当前配置
函数原型	void　USBDevEndpointConfigGet（uint32_t　ui32Base, uint32_t ui32Endpoint, uint32_t　* pui32MaxPacketSize, uint32_t　* pui32Flags）
参数	ui32Base　指定 USB 模块基地址 ui32Endpoint　待访问的端点 pui32MaxPacketSize　指向写入该端点的最大数据包大小的指针 pui32Flags　指向写入当前端点设置的指针。进入该函数，该指针必须包含 USB_EP_DEV_IN 或 USB_EP_DEV_OUT 之一，以指示查询的是 IN 端点还是 OUT 端点
描述	该函数返回从机模式下一个端点的基本配置。这个端点在 pulMaxPacketSize 和 pulFlags 返回的值，相当于 ulMaxPacketSize 和 pulFlags 之前传递给 USBDevEndpointConfigSet( )的值
返回	无

表 L-9　USBDevEndpointConfigSet

功能	设置一个端点的配置
函数原型	void　USBDevEndpointConfigSet（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32MaxPacketSize, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块基地址 ui32Endpoint　待访问的端点 ui32MaxPacketSize　端点的最大数据包大小 ui32Flags　用于配置其他断点的设置
描述	该函数设置在从机模式下端点的基本配置。由于端点 0 无动态配置，所以该函数不能被端点 0 调用。参数 ui32flags 可决定某些参数的配置，而其他参数提供了剩余部分的配置 ➢ USB_EP_MODE_标志　　//用于定义给定端点的类型 ➢ USB_EP_MODE_CTRL　　//控制端点 ➢ USB_EP_MODE_ISOC　　//等时端点 ➢ USB_EP_MODE_BULK　　//批量端点

（续）

参数	➢ USB_EP_MODE_INT　　//中断端点 USB_EP_DMA_MODE_标志用于确定 DMA 访问端点数据 FIFO 的类型。可根据 DMA 控制器的配置和使用来选择 DMA 的模式。更多的 DMA 配置信息请参阅“使用 USB 与 UDMA 控制器”一节 在配置 IN 端点时，当 ui32MaxPacketSize 个数据字节一经写入 FIFO 中，指定的 USB_EP_AUTO_SET 位将立即启动 USB 总线上该端点的数据自动传输。该选项通常用于在无 DMA 交互式要求时启动数据传输 在配置一个 OUT 端点时，一旦 FIFO 有足够的空间能接收超过 ui32MaxPacketSize 个字节的数据，指定的 USB_EP_AUTO_REQUEST 位将触发接收更多的数据请求。另外，对于 OUT 端点，一旦从 FIFO 读取数据，USB_EP_AUTO_CLEAR 位可用来自动清除数据包准备好标志。如果不使用此选项，此标志必须通过调用 USBDevEndpointStatusClear（）函数来手动清除。这两种设置可用于在 DMA 模式下消除额外的调用需求
返回	无

表 L-10　USBDevEndpointDataAck

功能	在设备模式下对从给定端点的 FIFO 中读取数据的应答
函数原型	void　USBDevEndpointDataAck（uint32_t ui32Base, uint32_t ui32Endpoint, bool bIsLastPacket）
参数	ui32Base　指定 USB 模块基地址 ui32Endpoint　待访问的端点 bIsLastPacket　指示是否为最后一个数据包
描述	该函数应答从端点的 FIFO 中读取的数据。当参数 bIsLastPacket 设置为 true 时，数据包为端点 0 上连续数据包中的最后一个。参数 bIsLastPacket 不能用于除端点 0 外的其他端点。在需要处理读取数据和应答已读取的数据时，可调用这个函数
返回	无

表 L-11　USBDevEndpointStall

功能	在设备模式下停止指定的端点
函数原型	void　USBDevEndpointStall（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块基地址 ui32Endpoint　待访问的端点 ui32Flags　指定停止的是 IN 端点还是 OUT 端点
描述	该函数使端点号传递进入停止条件。若参数 ui32Flags 是 USB_EP_DEV_IN，则停止这个端点的 IN 部分。如果参数 ui32Flags 是 USB_EP_DEV_OUT，则将停止这个端点的 OUT 部分
返回	无

表 L-12　USBDevEndpointStallClear

功能	在设备模式下清除指定端点上的停止条件
函数原型	void　USBDevEndpointStallClear（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块基地址 ui32Endpoint　待访问的端点 ui32Flags　指定要清除的中止条件是 IN 端点还是 OUT 端点

（续）

描述	该函数使端点号传递到退出停止条件。如果参数 ui32Flags 为 USB_EP_DEV_IN，则停止条件将从这个端点的 IN 部分删除。若参数 ui32Flags 为 USB_EP_DEV_OUT，则停止条件将从这个端点的 OUT 部分删除
返回	无

表 L-13　USBDevEndpointStatusClear

功能	在设备模式中清除该端点的状态位
函数原型	Void　USBDevEndpointStatusClear（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块基地址 ui32Endpoint　待访问的端点 ui32Flags　指定要清除的中止条件是 IN 端点还是 OUT 端点
描述	该函数可清除传递给参数 ui32Flags 的任何状态位。参数 ui32Flags 可获取调用 USBEndpointStatus( )的返回值
返回	无

表 L-14　USBDevMode

功能	将 USB 控制器的模式变更为从机模式
函数原型	void　USBDevMode（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块基地址
描述	该函数将 USB 控制器的模式改变成设备模式
返回	无

表 L-15　USBDevSpeedGet

功能	返回在设备模式下 USB 控制器的当前速度
函数原型	uint32_t　USBDevSpeedGet（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块基地址
描述	该函数返回连接到 USB 主机控制器的运行速度。这个函数返回 USB_HIGH_SPEED 或 USB_FULL_SPEED 之一来指示从机模式下的连接速度
返回	要么返回 USB_HIGH_SPEED，要么返回 USB_FULL_SPEED

表 L-16　USBEndpointDataAvail

功能	确定给定端点的 FIFO 中可用的数据的字节的数量
函数原型	uint32_t USBEndpointDataAvail（uint32_t ui32Base, uint32_t ui32Endpoint）
参数	ui32Base　指定 USB 模块基地址 ui32Endpoint　待访问的端点
描述	该函数返回给定接收（OUT）端点的 FIFO 中当前可用数据的字节数。它可用于先期调用 USBEndpointDataGet（）来确定存放新接收的数据包所需缓冲区的大小
返回	这个调用返回给定端点的 FIFO 中可用的字节数

表 L-17　USBEndpointDataGet

功能	获取给定端点 FIFO 中的数据
函数原型	int32_t USBEndpointDataGet（uint32_t ui32Base, uint32_t ui32Endpoint, uint8_t ＊pui8Data, uint32_t ＊pui32Size）
参数	ui32Base　指定 USB 模块基地址 ui32Endpoint　待访问的端点 pui8Data　指向用于从 FIFO 中返回数据的数据区指针 pui32Size　通过参数 pui8Data 传递给这个调用的初始化缓冲区大小。它被设置为在 FIFO 中返回的数据个数
描述	该函数返回给定端点 FIFO 中的数据。参数 pui32Size 表示传递给参数 pui32Data 的缓冲区大小。改变参数 pui32Size 中的数据，以匹配参数 pui8Data 中返回数据的个数。如果接收一个 0 字节的数据包，这个调用虽不会返回一个错误，但会在 pui32Size 参数中返回一个 0。当没有可用的数据包时，才会产生唯一的错误情况
返回	这个调用将返回 0；如果返回 -1，则没有收到数据包

表 L-18　USBEndpointDataPut

功能	将数据存放到给定端点的 FIFO 中
函数原型	int32_t USBEndpointDataPut（uint32_t ui32Base, uint32_t ui32Endpoint, uint8_t ＊pui8Data, uint32_t ＊pui32Size）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 pui8Data　指向作为存放到 FIFO 中数据源的数据区指针 pui32Size　返回存放到 FIFO 中的数据个数
描述	该函数把参数 pui8Data 中的数据存放到该端点的 FIFO 中。如果一个数据包已经等待传输，则该调用不会把任何数据存放到 FIFO 中，并返回 -1。注意不能写入超过 USBFIFO-ConfigSet( )分配给 FIFO 所能保存的数据
返回	返回 0 表示调用成功；返回 -1，表示 FIFO 处于使用中不能写入

表 L-19　USBEndpointDataSend

功能	开始一个端点的 FIFO 中的数据传输
函数原型	int32_t　USBEndpointDataSend（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32TransType）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 ui32TransType　指示待发送数据的类型
描述	该函数开始一个给定端点的 FIFO 数据传输。如果该端点的 USB_EP_AUTO_SET 位未被使能，则会调用这个函数。设置待请求的事务类型的参数 ui32TransType 允许在 USB 总线上出现适当的信号 ui32TransType 参数必须为下列值之一： ➢ USB_TRANS_OUT　//在主机模式下任何端点的 OUT 事务 ➢ USB_TRANS_IN　//在设备模式下任何端点的 IN 事务 ➢ USB_TRANS_IN_LAST　//在 IN 事务序列中端点 0 上的最后 IN 事务 ➢ USB_TRANS_SETUP　//设置端点 0 上的事务 ➢ USB_TRANS_STATUS　//端点 0 的状态结果
返回	返回 0 表示调用成功；如果传输已在进行中则返回 -1

表 L-20　USBEndpointDataToggleClear

功能	将一个端点上的数据反转设置成 0
函数原型	void USBEndpointDataToggleClear（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　指定端点复位数据反转 ui32Flags　指示待访问的是 IN 端点还是 OUT 端点
描述	该函数用于 USB 控制器清除一个端点的数据反转。该调用可以用在作为主机或从机的控制器中，但对端点 0 无效 参数 ui32Flags 必须为下列值中的一个： ➢ USB_EP_HOST_OUT ➢ USB_EP_HOST_IN ➢ USB_EP_DEV_OUT ➢ USB_EP_DEV_IN
返回	无

表 L-21　USBEndpointDMAChannel

功能	设置用于给定端点的 DMA 通道
函数原型	void　USBEndpointDMAChannel（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Channel）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　指定返回哪个端点的 FIFO 地址 ui32Channel　指定 DMA 通道使用哪个端点
描述	该函数用来配置给定端点的 DMA 通道。接收 DMA 通道只能用于接收终端，同理，发送 DMA 通道也只能用于发送端点。因此，3 个接收和发送 DMA 通道可以被映射到除 0 之外的任何端点。传递到 ui32Channel 中的值即是在 udma. h 中定义的 UDMA_CHANNEL_USBEP * values 的值
返回	无

表 L-22　USBEndpointDMAConfigSet

功能	配置 DMA 的端点设置
函数原型	void　USBEndpointDMAConfigSet（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Config）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 ui32Channel　指定端点的配置选项
描述	该函数配置一个给定端点的 DMA 设置，而不改变已配置的其他选项。为了使能 DMA 传输，在 DMA 传输之前必须先行调用 USBEndpointDMAEnable( ) 函数。配置选项传递到参数 ui32Config 中，其值的描述如下 用下列值之一来指定方向： ➢ USB_EP_HOST_OUT 或 USB_EP_DEV_IN　　　//设置从内存到 USB 的控制器 DMA 传输 ➢ USB_EP_HOST_IN 或 USB_EP_DEV_OUT//设置从 USB 控制器到内存的 DMA 传输

（续）

参数	用下列值之一来指定模式： ➢ USB_EP_DMA_MODE_0（default）//设置通常用于不跨越多个数据包或当每个数据包需要中断时的传输 ➢ USB_EP_DMA_MODE_1//设置通常用于跨越多个数据包和包之间不需要中断的传输 仅用于 USB_EP_HOST_OUT 或 USB_EP_DEV_IN 的值 ➢ USB_EP_AUTO_SET//设置当一个完整的数据包加载到 FIFO 时，允许发送 DMA 传输自动发送。这需用 USB_EP_DMA_MODE_1 确保当 FIFO 已满时将数据包发送出去，以及 DMA 能发送更多的数据。 仅用于 USB_EP_HOST_IN 或 USB_EP_DEV_OUT 的值： ➢ USB_EP_AUTO_CLEAR//设置在接收时允许接收 DMA 传输自动应答。这需用 usb_ep_dma_mode_1 确保当 FIFO 被 DMA 传输清空时，数据包能继续被接受和应答 仅用于 USB_EP_HOST_IN 的值： ➢ USB_EP_AUTO_REQUEST//设置当前次传输已清空的 FIFO 时，允许接收 DMA 传输自动发送一个新的 IN 事务请求。这通常用于与 USB_EP_AUTO_CLEAR 结合，以便接收 DMA 传输可以继续进行而不中断主处理器
返回	无

表 L-23　USBEndpointDMADisable

功能	禁止给定端点的 DMA
函数原型	void　USBEndpointDMADisable（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 ui32Flags　指定禁止那个方向
描述	该个函数禁止一个给定端的 DMA，以允许无 DMA USB 事务能正常产生中断。参数 ui32Flags 必须为 USB_EP_DEV_IN 或 USB_EP_DEV_OUT；而所有其他位将被忽略
返回	无

表 L-24　USBEndpointDMAEnable

功能	使能一个给定端点的 DMA
函数原型	void　USBEndpointDMAEnable（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 ui32Flags　在使能 DMA 时指定使用那个方向和那种模式
描述	该函数使能一个给定端点的 DMA，以及根据参数 ui32Flags 中的值来配置其模式。参数 ui32Flags 必须使 USB_EP_DEV_IN 或 USB_EP_DEV_OUT 置位。一旦调用这个函数，唯一的 DMA 或错误中断将由 USB 控制器产生
返回	无

表 L-25　USBEndpointPacketCountSet

功能	设置在传输多个批量包时请求包的个数
函数原型	void　USBEndpointPacketCountSet（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Count）

（续）

参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 ui32Flags　请求包个数
描述	该函数在使用 DMA 传输多个批量数据包时用于设置请求连续批量包的个数
返回	无

表 L-26　USBEndpointStatus

功能	返回端点的当前状态
函数原型	uint32_t　USBEndpointStatus（uint32_t　ui32Base， uint32_t ui32Endpoint）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点
描述	该函数返回一个给定的端点的状态。如果必须要清除这些状态位，则必须调用 USBDevEndpointStatusClear( ) 或 USBHostEndpointStatusClear( ) 函数 下面为主机模式下的状态标志： ➢ USB_HOST_IN_PID_ERROR//在给定端点上出现 PID 错误 ➢ USB_HOST_IN_NOT_COMP//设备对 IN 请求响应失败 ➢ USB_HOST_IN_STALL//收到一个 IN 端点的停止信号 ➢ USB_HOST_IN_DATA_ERROR//在同步模式下，IN 端点出现一个 CRC 或位填充错误 ➢ USB_HOST_IN_NAK_TO//该 IN 端点接收到的 NAK 时间超过了指定的超时周期 ➢ USB_HOST_IN_ERROR//使用该 IN 端点与一个设备通信失败 ➢ USB_HOST_IN_FIFO_FULL//该 IN 端点的 FIFO 已满 ➢ USB_HOST_IN_PKTRDY//该 IN 端点的数据包已准备就绪 ➢ USB_HOST_OUT_NAK_TO//该 OUT 端点接收到的 NAK 时间超过了指定的超时周期 ➢ USB_HOST_OUT_NOT_COMP//设备对 OUT 请求响应失败 ➢ USB_HOST_OUT_STALL//该输 OUT 端点上收到一个停止信号 ➢ USB_HOST_OUT_ERROR//使用该 OUT 端点与一个设备通信失败 ➢ USB_HOST_OUT_FIFO_NE//该端点的输出 FIFO 为非空 ➢ USB_HOST_OUT_PKTPEND//该 OUT 端点的数据传输没有完成 ➢ USB_HOST_EP0_NAK_TO//端点 0 接收到的 NAK 时间超过了指定的超时周期 ➢ USB_HOST_EP0_ERROR//设备响应端点 0 的请求失败 ➢ USB_HOST_EP0_IN_STALL//在一次 IN 传输中，端点 0 收到一个停止信号 ➢ USB_HOST_EP0_IN_PKTRDY//端点 0 上的数据包已为一次 IN 传输准备就绪 下面为设备模式下的状态标志： ➢ USB_DEV_OUT_SENT_STALL//该 OUT 端点发送了一个停止信号 ➢ USB_DEV_OUT_DATA_ERROR//在一个 OUT 端点上出现了一个 CRC 或位填充错误 ➢ USB_DEV_OUT_OVERRUN//由于 FIFO 已满，使输出包无法加载 ➢ USB_DEV_OUT_FIFO_FULL// OUT 端点的 FIFO 已满 ➢ USB_DEV_OUT_PKTRDY// OUT 端点的 FIFO 中有一个准备就绪的数据包 ➢ USB_DEV_IN_NOT_COMP//一个较大的包已被拆分，使更多的数据传入 ➢ USB_DEV_IN_SENT_STALL// 在该 IN 端点上发送一个停止信号 ➢ USB_DEV_IN_UNDERRUN// 在该 IN 端点上请求数据并且无数据准备好 ➢ USB_DEV_IN_FIFO_NE// IN 端点的 FIFO 非空 ➢ USB_DEV_IN_PKTPEND//该 IN 端点的数据传输还未完成 ➢ USB_DEV_EP0_SETUP_END//在发送数据结束条件前，一个控制传输已结束 ➢ USB_DEV_EP0_SENT_STALL//端点 0 上发送了一个停止信号 ➢ USB_DEV_EP0_IN_PKTPEND//端点 0 上的数据传输还未完成 ➢ USB_DEV_EP0_OUT_PKTRDY//在端点 0 上的输出 FIFO 有一个数据包已准备就绪
返回	取决于模式的端点当前状态标志

表 L-27　USBFIFOAddrGet

功能	返回一个给定端点的绝对 FIFO 地址
函数原型	uint32_t USBFIFOAddrGet（uint32_t ui32Base， uint32_t ui32Endpoint）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　指定待返回的端点的 FIFO 地址
描述	该函数返回 FIFO 的实际物理地址。USB 需要该地址来进行 μDMA 操作，并且源地址或目标地址也必须设置成一个给定端点的物理 FIFO 地址
返回	无

表 L-28　USBFIFOConfigGet

功能	返回端点的 FIFO 配置
函数原型	void　USBFIFOConfigGet（uint32_t ui32Base， uint32_t ui32Endpoint， uint32_t ＊pui32FIFOAddress， uint32_t ＊pui32FIFOSize， uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 pui32FIFOAddress　FIFO 的起始地址 pui32FIFOSize　由 USB_FIFO_SZ_ values 的值之一来指定 FIFO 的大小 ui32Flags　指定从 FIFO 配置中获取的信息
描述	该函数返回一个给定端点 FIFO 的起始地址和大小。由于端点 0 不能动态配置 FIFO，所以该函数不能被端点 0 调用 参数 ui32Flags 指定读取端点为 OUT FIFO 和 IN FIFO 之一。如果在主机模式下，ui32Flags 参数为 USB_EP_HOST_OUT 或 USB_EP_HOST_IN；如果在设备模式下，则 ui32Flags 参数必须为 USB_EP_DEV_OUT 或 USB_EP_DEV_IN
返回	无

表 L-29　USBFIFOConfigSet

功能	设置端点的 FIFO 配置
函数原型	void　USBFIFOConfigSet（uint32_t ui32Base， uint32_t ui32Endpoint， uint32_t ui32FIFOAddress， uint32_t ui32FIFOSize， uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 pui32FIFOAddress　FIFO 的起始地址 pui32FIFOSize　由 USB_FIFO_SZ_ values 中的值之一来指定 FIFO 的大小 ui32Flags　指定从 FIFO 配置中获取的信息
描述	该函数配置一个给定端点的起始 FIFO RAM 地址和 FIFO 大小。由于端点 0 不能动态配置 FIFO，所以该函数不能被端点 0 调用。参数 ui32FIFOSize 必须为 USB_FIFO_SZ_values 中的值之一 ui32FIFOAddress 的值必须是 8 字节的倍数，并直接表示 USB 控制器 FIFO RAM 中的起始地址 参数 ui32Flags 指定读取端点为 OUT FIFO 和 IN FIFO 之一。如果在主机模式下，ui32Flags 参数为 USB_EP_HOST_OUT 或 USB_EP_HOST_IN；如果在设备模式下，则 ui32Flags 参数必须为 USB_EP_DEV_OUT 或 USB_EP_DEV_IN
返回	无

表 L-30　USBFIFOFlush

功能	强制刷新端点的 FIFO
函数原型	void　USBFIFOFlush（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 ui32Flags　指定待访问的是 IN 端点，还是 OUT 端点
描述	该函数强制刷新 USB 控制器 FIFO 中的数据。该函数既可被主机调用，也可以被设备控制器（从机）所调用。 参数 ui32Flags 的取值为下值之一： ➢ USB_EP_HOST_IN ➢ USB_EP_HOST_OUT ➢ USB_EP_DEV_OUT ➢ USB_EP_DEV_IN
返回	无

表 L-31　USBFrameNumberGet

功能	获取当前帧号
函数原型	uint32_t USBFrameNumberGet（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块的基地址
描述	该函数返回接收到的最后一帧的帧号
返回	收到的最后一帧号的帧号

表 L-32　USBHighSpeed

功能	使能或禁止 USB 高速协商（negotiation）
函数原型	void　USBHighSpeed（uint32_t ui32Base，bool bEnable）
参数	ui32Base　指定 USB 模块的基地址 bEnable　指定是使能高速协商还是禁止高速协商
描述	当调用函数的参数 bEnable 设置为 true 时，均可使能主机和设备模式的高速协商。在设备模式下当 USB 控制器从主机接收一个复位信号时会引起设备的高速协商。在主机模式下当所连接设备复位时 USB 主机将使能高速协商。如果 bEnable 设置为 false 控制器只在全速或低速下运行
返回	返回端点当前正在使用的功能地址

表 L-33　USBHostAddrGet

功能	获取端点当前功能设备的地址
函数原型	uint32_t　USBHostAddrGet（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 ui32Flags　确定是 IN 端点，还是 OUT 端点
描述	该函数返回与设备通信端点的功能地址。参数 ui32Flags 确定是返回 IN 端点的设备地址，还是返回 OUT 端点的设备地址
返回	返回当前正在使用端点的功能地址

表 L-34　USBHostAddrSet

功能	设置在主机模式下连接到一个端点设备的功能地址
函数原型	void　USBHostAddrSet（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Addr, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 ui32Addr　控制器使用该端点的功能地址 ui32Flags　确定是 IN 端点，还是 OUT 端点
描述	该函数配置使用该端点进行通信设备的功能地址。参数 ui32Addr 是与这个端点进行通信的目标设备的地址。参数 ui32Flags 指示是要设置 IN 端点，还是 OUT 端点
返回	无

表 L-35　USBHostEndpointConfig

功能	设置主机端点的基本配置
函数原型	void　USBHostEndpointConfig（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32MaxPayload, uint32_t ui32NAKPollInterval, uint32_t ui32TargetEndpoint, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 ui32MaxPayload　该端点的最大有效载荷 ui32NAKPollInterval　根据端点类型的 NAK 超时限制或轮询间隔 ui32TargetEndpoint　主机端点的目标端点 ui32Flags　用于配置其他端点设置
描述	该函数设置主机模式中的一个端点发送或接收部分的基本配置。参数 ui32Flags 确定一些配置而剩余的配置可由其他参数给出。参数 ui32Flags 决定是一个 IN 端点（USB_EP_HOST_IN 或 USB_EP_DEV_IN），还是 OUT 的端点（USB_EP_HOST_OUT 或 USB_EP_DEV_OUT）。并且决定是一个全速端点（USB_EP_SPEED_FULL），还是低速（USB_EP_SPEED_LOW）端点 USB_EP_MODE_flags 控制端点的类型： ➢ USB_EP_MODE_CTRL　//控制端点 ➢ USB_EP_MODE_ISOC　//等时端点 ➢ USB_EP_MODE_BULK　//批量端点 ➢ USB_EP_MODE_INT　//中断端点 根据 USB_EP_MODE 的值和端点 0 以及另外的端点是否调用该函数，将使参数 ui32NAKPollInterval 的含义不同。对于端点 0 或批量端点，该值总是表示在认为超时之前允许设备 NAK 的帧数。如果是同步端点或中断端点，则该值为这个端点的轮询间隔 对于中断端点，轮询间隔为端点 IN 请求中断之间的帧数且其范围为 1 ~ 255。对于同步端点，此值表示轮询间隔为 2^（ui32NAKPollInterval － 1）的帧。在用于 NAK 超时，ui32NAKPollInterval 的值在发布一个超时前被指定为 2^（ui32NAKPollInterval － 1）的帧 当设置 ui32nakpollinterval 值时，可指定两个特殊的超时值。其一为 MAX_NAK_LIMIT，它是传递给该变量的最大值；其二为 DISABLE_NAK_LIMIT，它表明没有 NAK 的数量限制 USB_EP_DMA_MODE_flags 使能 DMA 类型用来访问端点的数据 FIFO。DMA 的模式选择取决于如何配置 DMA 控制器及其用途

描述	在配置端点的 OUT 部分时，当由 ui32MaxPayload 指定的字节数一经写入到该端点的 OUT FIFO 中时，则可指定 USB_EP_AUTO_SET 位来启动 USB 总线上的数据传输 在配置端点的 IN 部分时，一旦 FIFO 腾出足够的空间以适应 ui32MaxPayload 字节，可指定 USB_EP_AUTO_REQUEST 位以触发更多的数据请求。一旦从 FIFO 中读取数据完成，可用 USB_EP_AUTO_CLEAR 位来自动清除数据包的准备就绪标志。如果不使用此选项，则必须通过调用 USBDevEndpointStatusClear( ) 或 USBHostEndpointStatusClear( ) 函数来手工清除该标志
返回	无

表 L-36　USBHostEndpointDataAck

功能	在主机模式下对从给定端点 FIFO 中读取数据的应答
函数原型	void　USBHostEndpointDataAck（uint32_t ui32Base, uint32_t ui32Endpoint）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点
描述	该函数为从端点的 FIFO 中读取数据的应答。如果需要在读取数据与应答已读取的数据之间进行处理，需调用这个函数
返回	无

表 L-37　USBHostEndpointDataToggle

功能	在主机模式下设置端点的翻转数据值
函数原型	Void　USBHostEndpointDataToggle（uint32_t ui32Base, uint32_t ui32Endpoint, bool　bDataToggle, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　指定复位数据翻转的端点 bDataToggle　指定状态是设置为 DATA0，还是 DATA1 ui32Flags　指定是 IN 端点，还是 OUT 端点
描述	该函数用于在主机模式下强制数据状态的翻转。如果传递给参数 bDataToggle 的值为 false，则数据翻转设置为 DATA0 状态，若为 true，则成 DATA1 状态。参数 ui32Flags 可为 USB_EP_HOST_IN 或 USB_EP_HOST_OUT 用于分别访问该端点所需的部分。而端点 0 忽略 ui32Flags 参数
返回	无

表 L-38　USBHostEndpointPing

功能	在主机模式下对于使用高速控制传输的端点使能或禁止 ping 令牌
函数原型	void　USBHostEndpointPing（uint32_t ui32Base, uint32_t ui32Endpoint, bool bEnable）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　指定是使能还是禁止 ping 令牌的端点 bEnable　指定是使能还是禁止 ping 令牌
描述	该函数在数据和状态阶段的高速控制传输期间用于配置 USB 控制器以发送或者不发送 ping 令牌。ui32Endpoint 仅支持 USB_EP_0，因为处理所有的控制传输都使用这个端点。如果 bEnable 为真，则使能 ping 令牌；如果 bEnable 为 false，则禁止 ping 令牌。不支持设备在使用高速模式时 ping 令牌
返回	无

表 L-39　USBHostEndpointSpeed

功能	更改主机端点连接的速度
函数原型	void　USBHostEndpointSpeed（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 ui32Flags　用于配置其他端点的设置
描述	该函数用于在主机模式下设置 IN 或 OUT 端点的 USB 速度。参数 ui32Flags 可使用下列值之一来指定速度： ➢ USB_EP_SPEED_LOW ➢ USB_EP_SPEED_FULL ➢ USB_EP_SPEED_HIGH 参数 ui32Flags 还可通过添加 USB_EP_HOST_IN 或 USB_EP_HOST_OUT 的逻辑或来指定方向的设置
返回	无

表 L-40　USBHostEndpointStatusClear

功能	清除在主机模式下该端点的状态位
函数原型	void　USBHostEndpointStatusClear（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 ui32Flags　待清除的状态位
描述	该函数清除传递给参数 ui32Flags 任何位的状态。参数 ui32Flags 为 USBEndpointStatus( ) 调用的返回值
返回	无

表 L-41　USBHostHubAddrGet

功能	获取该端点当前设备的 HUB（集线器）地址
函数原型	uint32_t　USBHostHubAddrGet（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点 ui32Flags　确定是 IN 端点，还是 OUT 端点
描述	该函数返回当前正在使用的与设备通信端点的 HUB 地址。参数 ui32Flags 决定返回的是 IN 端点的设备地址，还是 OUT 端点的设备地址 注意：这个函数只能在主机模式下调用
返回	该函数返回当前正在使用的一个端点的 HUB 地址

表 L-42 USBHostHubAddrSet

功能	设置连接到一个端点的设备 HUB 地址
函数原型	void USBHostHubAddrSet（uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Addr, uint32_t ui32Flags）
参数	ui32Base 指定 USB 模块的基地址 ui32Endpoint 待访问的端点 ui32Addr 使用该端点的设备 HUB 地址和端口，且必须用 0 ~ 6 位定义 HUB 地址，而用 8 ~ 14 位定义端口号 ui32Flags 确定是 IN 端点，还是 OUT 端点
描述	该函数配置与正在使用的端点进行通信设备的 HUB 地址。参数 ui32Flags 决定这个调用是配置 IN 端点的设备地址，还是配置 OUT 端点的设备地址，并设置下游设备的速度。有效的值为 USB_EP_HOST_OUT 或 USB_EP_HOST_IN 其中之一个与 USB_EP_SPEED_LOW 的"或运算"
返回	无

表 L-43 USBHostMode

功能	变更 USB 控制器到主机的模式
函数原型	void USBHostMode（uint32_t ui32Base）
参数	ui32Base 指定 USB 模块的基地址
描述	该函数将把 USB 控制器模式改变为主机模式
返回	无

表 L-44 USBHostPwrConfig

功能	设置 USB 电源故障的配置
函数原型	void USBHostPwrConfig（uint32_t ui32Base, uint32_t ui32Flags）
参数	ui32Base 指定 USB 模块的基地址 ui32Flags 指定电源故障的配置
描述	该函数控制 USB 控制器如何使用它的外部电源控制的引脚（USBnPFLT、USBnEPEN）。标志用来指定电源故障电平的灵敏度、电源故障动作，电源使能电平和来源 可选择下列其中一种作为电源故障的电平灵敏度： ➢ USB_HOST_PWRFLT_LOW//拉低引脚电平来指示外部电源故障 ➢ USB_HOST_PWRFLT_HIGH//拉高引脚电平来指示外部电源故障 可选择下列其中的一种作为电源故障的动作： ➢ USB_HOST_PWRFLT_EP_NONE//当检测到电源故障时无自动操作 ➢ USB_HOST_PWRFLT_EP_TRI//当电源故障时 USBnEPEN 引脚将自动变成三态 ➢ USB_HOST_PWRFLT_EP_LOW//在电源故障时 USBnEPEN 引脚将自动被驱动为低电平 ➢ USB_HOST_PWRFLT_EP_HIGH//在电源故障时 USBnEPEN 引脚将自动被驱动为高电平 可选择下列操作中的一种作为电源的等级使能和来源： ➢ USB_HOST_PWREN_MAN_LOW//当调用 USBHostPwrEnable( )时，USBnEPEN 引脚将由 USB 控制器驱动为低电平 ➢ USB_HOST_PWREN_MAN_HIGH//当调用 USBHostPwrEnable( )时，USBnEPEN 引脚将由 USB 控制器驱动为高电平 ➢ USB_HOST_PWREN_AUTOLOW//如果 USBOTGSessionRequest( )已启用了会话，USBnEPEN 引脚将由 USB 控制器自动驱动为低电平 ➢ USB_HOST_PWREN_AUTOHIGH//如果 USBOTGSessionRequest( )已启用了会话，USBnEPEN 引脚将由 USB 控制器自动驱动为高电平 在支持 VBUS 故障滤波器的设备上，可以添加 USB_HOST_PWREN_FILTER 来消除由高功耗引起的小的、短促的 VBUS 电平下降。此特性主要用于避免由于设备的高浪涌电流造成的 VBUS 错误
返回	无

表 L-45　USBHostPwrDisable

功能	禁止外部电源引脚
函数原型	void　USBHostPwrDisable（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块的基地址
描述	该函数禁止 USBnEPEN 信号，禁止外部电源在主机模式中操作
返回	无

表 L-46　USBHostPwrEnable

功能	使能外部电源引脚
函数原型	void　USBHostPwrEnable（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块的基地址
描述	该函数使能 USBnEPEN 信号，使能外部电源在主机模式中操作
返回	无

表 L-47　USBHostPwrFaultDisable

功能	禁止电源故障检测
函数原型	void　USBHostPwrFaultDisable（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块的基地址。
描述	该函数禁止 USB 控制器的电源故障检测
返回	无

表 L-48　USBHostPwrFaultEnable

功能	使能电源故障检测
函数原型	void　USBHostPwrFaultEnable（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块的基地址
描述	该函数使能 USB 控制器中的电源故障检测。如果在 USBnPFLT 引脚不使用时，该函数不能被使用
返回	无

表 L-49　USBHostRequestIN

功能	在主机模式中预定一个端点上的 IN 事务请求
函数原型	void　USBHostRequestIN（uint32_t ui32Base, uint32_t ui32Endpoint）
参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点
描述	该函数预定一个 IN 事务请求。在通信的 USB 设备响应数据时，可以通过调用 USBEndpointDataGet( )或通过 DMA 传输来获取数据
返回	无

表 L-50　USBHostRequestINClear

功能	清除主机模式中端点的一个预定 IN 事务
函数原型	void　USBHostRequestINClear（uint32_t ui32Base, uint32_t ui32Endpoint）

参数	ui32Base　指定 USB 模块的基地址 ui32Endpoint　待访问的端点
描述	如果当前仍被挂起，该函数用于清除之前预定的 IN 事务。如果由 ui32Endpoint 指定的端点被重新配置成与其他设备进行通信，该函数被用来安全地禁用任何预定 IN 事务
返回	无

表 L-51　USBHostRequestStatus

功能	在端点 0 上发布一个状态 IN 事务请求
函数原型	void　USBHostRequestStatus（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块的基地址
描述	该函数用来发布端点 0 上设备的状态 IN 事务请求。该函数只能与端点 0 一起使用，这是因为只有控制端点支持这种功能。此函数用于完成一个设备最后阶段的控制事务且当收到状态数据包时会发出一个中断信号
返回	无

表 L-52　USBHostReset

功能	处理 USB 总线的复位条件
函数原型	void　USBHostReset（uint32_t ui32Base, bool bStart）
参数	ui32Base　指定 USB 模块的基地址 bStart　指定在 USB 总线上发出的是启动复位信号，还是停止复位信号
描述	在该函数被调用且参数 bStart 设置为 true 时，该函数将启动 USB 总线上的复位条件。调用者必须延迟 20ms 以上才能再次调用参数 bStart 设置为 false 的函数
返回	无

表 L-53　USBHostResume

功能	处理 USB 总线的恢复条件
函数原型	void　USBHostResume（uint32_t ui32Base, bool bStart）
参数	ui32Base　指定 USB 模块的基地址 bStart　指定 USB 控制器是进入恢复信号状态，还是离开恢复信号状态
描述	在设备模式时，该函数使 USB 控制器脱离挂起状态。此调用必须先把参数 bStart 设置为 true 来启动恢复信号。然后，在设备应用程序必须延迟至少 10ms 但不超过 15 ms 后，方可调用参数 bStart 设置为 false 的函数( )。 在主机模式时，这个函数发出设备离开挂起状态的信号。此调用必须先把参数 bStart 设置为 true 来启动恢复信号。然后，在主机应用程序必须延迟至少 20ms 后，才可调用参数 bStart 设置为 false 的函数( ) 这个操作将使控制器完成在 USB 总线上的恢复信号
返回	无

表 L-54　USBHostSpeedGet

功能	返回当前被连接的 USB 设备的速度
函数原型	uint32_t USBHostSpeedGet（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块的基地址
描述	该函数返回在主机模式中当前 USB 总线的速度

（续）

返回	返回下列值之一： ➢ USB_LOW_SPEED ➢ USB_FULL_SPEED ➢ USB_UNDEF_SPEED

表 L-55　USBHostSuspend

功能	使 USB 总线处于挂起状态
函数原型	void　USBHostSuspend（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块的基地址
描述	在使用主机模式时，该函数使 USB 总线处于挂起状态
返回	无

表 L-56　USBIntDisableControl

功能	禁止给定 USB 控制器上的控制中断
函数原型	void　USBIntDisableControl（uint32_t ui32Base, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Flags　指定要禁止的控制中断
描述	该函数禁止由参数 ui32Base 指定的 USB 控制器的控制中断。参数 ui32Flags 指定要禁止的控制中断。传递到 ui32Flags 参数中的标志必须定义为以 USB_INTCTRL_ * 开始，而不是任何其他的 USB_INT 标志
返回	无

表 L-57　USBIntDisableEndpoint

功能	禁止给定 USB 控制器上的端点中断
函数原型	void　USBIntDisableEndpoint（uint32_t ui32Base, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Flags　指定要禁止的端点中断
描述	该函数禁止由参数 ui32Base 指定的 USB 控制器的端点中断。参数 ui32Flags 指定要禁止的端点中断。传递到 ui32Flags 参数中的标志必须定义为以 USB_INTEP__ * 开始，而不是任何其他的 USB_INT 标志
返回	无

表 L-58　USBIntEnableControl

功能	使能给定 USB 控制器上的控制中断
函数原型	void　USBIntEnableControl（uint32_t ui32Base, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Flags　指定要使能的控制中断
描述	该函数使能由参数 ui32Base 指定的 USB 控制器的控制中断。参数 ui32Flags 指定要使能的控制中断
返回	无

表 L-59　USBIntEnableEndpoint

功能	使能给定 USB 控制器上的端点中断
函数原型	void　USBIntEnableEndpoint（uint32_t ui32Base, uint32_t ui32Flags）
参数	ui32Base　指定 USB 模块的基地址 ui32Flags　指定要使能的端点中断
描述	该函数使能由参数 ui32Base 指定的 USB 控制器的端点中断。参数 ui32Flags 指定要使能的端点中断
返回	无

表 L-60　USBIntRegister

功能	注册 USB 控制器的中断处理程序
函数原型	void　USBIntRegister（uint32_t ui32Base, void（ * pfnHandler）（void）)
参数	ui32Base　指定 USB 模块的基地址 pfnHandler　当 USB 中断发生时指向被调用函数的指针
描述	该函数用于在 USB 中断发生时注册被调用的处理程序，并使能中断控制器中的 USB 全局中断。必须通过单独调用 USBIntEnable( ) 函数来使能指定所需的 USB 中断。由中断处理程序负责调用 USBIntStatusControl( ) 和 USBIntStatusEndpoint( ) 函数来清除中断源
返回	无

表 L-61　USBIntStatusControl

功能	返回给定 USB 控制器上的控制中断状态
函数原型	uint32_t　USBIntStatusControl（uint32_t　ui32Base）
参数	ui32Base　指定 USB 模块的基地址
描述	该函数读取一个 USB 控制器的控制中断状态。这个调用仅返回控制中断的当前状态，端点中断状态可通过调用 USBIntStatusEndpoint( ) 来获取。返回的位值与 USB_INCTRL_ * 值比较 以下是所有 USB_INCTRL_标志的含义及其有效模式。这些值适用于对任何 USBIntEnableControl( )、USBIntStatusControl( ) 和 USBIntDisableControl( ) 的调用。在下列模式中这些标志的唯一有效形式在括号中注明：Host、Device、OTG ➢ USB_INTCTRL_ALL//屏蔽所有控制中断源 ➢ USB_INTCTRL_VBUS_ERR//发生 VBUS 错误（仅主机） ➢ USB_INTCTRL_SESSION//在电缆 A 侧上的会话开始检测（仅 OTG） ➢ USB_INTCTRL_SESSION_END//会话结束检测（仅设备） ➢ USB_INTCTRL_DISCONNECT//设备断开检测（仅主机） ➢ USB_INTCTRL_CONNECT//设备连接检测（仅主机） ➢ USB_INTCTRL_SOF//起始帧检测 ➢ USB_INTCTRL_BABBLE//USB 控制器检测到设备发出的过去一帧的结束信号（仅主机） ➢ USB_INTCTRL_RESET//由设备发出的复位信号检测（仅设备） ➢ USB_INTCTRL_RESUME//恢复信号检测 ➢ USB_INTCTRL_SUSPEND//由设备发出的挂起信号检测（仅设备） ➢ USB_INTCTRL_MODE_DETECT//OTG 电缆模式检测已经完成（OTG） ➢ USB_INTCTRL_POWER_FAULT//电源故障检测（仅主机）
返回	返回 USB 控制器的控制中断状态

表 L-62　USBIntStatusEndpoint

功能	返回给定 USB 控制器的端点中断状态
函数原型	uint32_t USBIntStatusEndpoint（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块的基地址
描述	该函数读取 USB 控制器端点的中断状态。这个调用仅返回端点中断的当前状态，控制中断状态可通过调用 USBIntStatusControl( ) 来获取。返回的位值应该比对 USB_INTEP_ * 的值。这些值被分成 USB_INTEP_HOST_ * 和 USB_INTEP_DEV_ * 两组，以处理所有端点的主机和设备模式
返回	返回 USB 控制器的端点中断状态

表 L-63　USBIntUnregister

功能	注销 USB 控制器的中断处理程序
函数原型	void　USBIntUnregister（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块的基地址
描述	该函数注销中断处理程序。这个函数也可禁止中断控制器中的 USB 中断
返回	无

表 L-64　USBModeConfig

功能	改变 USB 控制器的操作模式
函数原型	Void　USBModeConfig（uint32_t ui32Base, uint32_t ui32Mode）
参数	ui32Base　指定 USB 模块的基地址 ui32Mode　指定的 USB OTG 引脚的操作模式
描述	该函数改变 USB 控制器的操作模式。在完全 OTG 模式下操作时，USB 控制器使用的 VBUS 和 ID 引脚来检测模式和电压的变化。而这些引脚主要用于 OTG 模式，它们也可以影响主机和设备模式的操作。在设备模式下 USB 控制器可配置成监控或忽略 VBUS。监控 VBUS 允许控制器确定它是否已从主机断开。在主机模式下，USB 控制器使用 VBUS 引脚，检测由于过度功耗引起的 VBUS 电压下降所带来的 VBUS 损失。这个调用取 USBHostMode( )、USBDeviceMode( )、USBOTGMode( )函数 ui32Mode 的值应为下列值之一： ➢ USB_MODE_OTG//使能完全 OTG 模式操作，VBUS 和 ID 由控制器使用 ➢ USB_MODE_HOST//使能仅作为主机的操作，无 VBUS 监控或 ID 引脚 ➢ USB_MODE_HOST_VBUS//使能仅作为主机的操作，有 VBUS 监控或 ID 引脚。这将使能 VBUS 下降检测，并强迫工作在主机模式下 ➢ USB_MODE_DEVICE//使能仅作为装置的操作，无 VBUS 监控或 ID 引脚 ➢ USB_MODE_DEVICE_VBUS/ /使能仅作为装置的操作带 VBUS 引脚监控。这将使能断开检测，并强迫工作在设备模式下
返回	无

表 L-65　USBModeGet

功能	返回控制器的当前操作模式
函数原型	uint32_t　USBModeGet（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块的基地址
描述	该函数返回 USB 控制器的当前操作模式 – OTG 或双模式功能 对于 OTG 控制器，该函数将返回 OTG 控制器的下值之一： ➢ USB_OTG_MODE_ASIDE_HOST ➢ USB_OTG_MODE_ASIDE_DEV

（续）

描述	➢ USB_OTG_MODE_BSIDE_HOST ➢ USB_OTG_MODE_BSIDE_DEV ➢ USB_OTG_MODE_NONE ➢ USB_OTG_MODE_ASIDE_HOST //指示在主机模式中控制器 A 侧电缆 ➢ USB_OTG_MODE_➢ ASIDE_DEV //指示在设备模式中控制器 A 侧电缆 ➢ USB_OTG_MODE_BSIDE_HOST //指示在主机模式中控制器 B 侧电缆 ➢ USB_OTG_MODE_BSIDE_DEV //指示在设备模式中控制器 B 侧电缆。如果一个 OTG 会话请求开始（在恰当的位置无电缆），这种模式为默认
描述	➢ USB_OTG_MODE_NONE //表明控制器不试图确定其在系统中的作用 对于双模式控制器，该函数将返回下列值之一： ➢ USB_DUAL_MODE_HOST ➢ USB_DUAL_MODE_DEVICE ➢ USB_DUAL_MODE_NONE ➢ USB_DUAL_MODE_HOST //指示控制器作为主机 ➢ USB_DUAL_MODE_DEVICE //指示控制器作为设备 ➢ USB_DUAL_MODE_NONE //表明控制器无论作为主机还是设备都未激活
返回	函数返回： ➢ USB_OTG_MODE_ASIDE_HOST ➢ USB_OTG_MODE_ASIDE_DEV ➢ USB_OTG_MODE_BSIDE_HOST ➢ USB_OTG_MODE_BSIDE_DEV ➢ USB_OTG_MODE_NONE ➢ USB_DUAL_MODE_HOST ➢ USB_DUAL_MODE_DEVICE ➢ USB_DUAL_MODE_NONE

表 L-66 USBNumEndpointsGet

功能	返回设备上 USB 端点对的个数
函数原型	uint32_t USBNumEndpointsGet（uint32_t ui32Base）
参数	ui32Base 指定 USB 模块的基地址
描述	该函数返回传递基地址对应的 USB 控制器支持的端点对个数。返回值为可用的 IN 或 OUT 端点的个数，但不包括端点 0（控制端点）。例如，如果返回 15，即除了端点 0 外，包括 15 个 IN 端点和 15 个 OUT 端点
返回	返回可用的 IN 或 OUT 端点的个数

表 L-67 USBOTGMode

功能	将 USB 控制器改变为 OTG 模式
函数原型	void USBOTGMode（uint32_t ui32Base）
参数	ui32Base 指定 USB 模块的基地址
描述	该函数用于将 USB 控制器改变为 OTG 模式。此函数只适用于具有 OTG 功能的微控制器
返回	无

表 L-68 USBOTGSessionRequest

功能	开始或结束一次会话
函数原型	void USBOTGSessionRequest（uint32_t ui32Base, bool bStart）
参数	ui32Base 指定 USB 模块的基地址 bStart 指定是调用开始会话，还是调用结束会话

（续）

描述	该函数用于在 OTG 模式下开始一次会话请求或结束一次会话。如果参数 bStart 设置为 true，则该函数将开始一次会话，如果 bStart 为 false 将结束一次会话
返回	无

表 L-69　USBPHYPowerOff

功能	关闭 USB PHY 电源
函数原型	void　USBPHYPowerOff（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块的基地址
描述	该函数可关闭 USB PHY 电源，以减少该设备的电流消耗。在关机状态 USB 控制器将无法操作
返回	无

表 L-70　USBPHYPowerOn

功能	打开 USB PHY 电源
函数原型	void　USBPHYPowerOn（uint32_t ui32Base）
参数	ui32Base　指定 USB 模块的基地址
描述	该函数打开 USB PHY 电源，使能它将返回正常操作状态。默认情况下，PHY 是打开的，所以只在已调用 USBPHYPowerOff( ) 函数时，才需调用该函数
返回	无

注：

1. Using USB with the uDMA Controller（固件库翻译省略，有需要用到这部分的读者请见 TI USB Controller 固件库部分）。

2. USB Link Power Management Functions（固件库翻译省略）。

3. USB UTMI Low Pin Interface（ULPI）（固件库翻译省略）。

参考文献

[1] TI spmu349, Tiva™ C Series TM4C123GH6PM ROM User Guide, 2013.

[2] TI SW－EK－TM4C123GXL－UG－2. 0. 1. 11577, 2013.

[3] TI Tiva™ TM4C123GH6PM Microcontroller Datasheet, 2013.

[4] TI SW－TM4C－DRL－UG－2. 0. 1. 11577, 2013.

[5] TI SW－TM4C－GRL－UG－2. 0. 1. 11577, 2013.

[6] TI SW－TM4C－USBL－UG－2. 0. 1. 11577, 2013.

[7] TI SW－DK－TM4C123G－UG－2. 0. 1. 11577, 2013.

[8] 广州周立功单片机发展有限公司. Stellaris 外设驱动库用户手册, 2008.

[9] Josseph Yiu. ARM Cortex－M3 权威指南［M］. 宋岩, 译. 北京：北京航空航天大学出版社, 2009.

[10] TI LM3S9B96 中文数据手册, 2011.

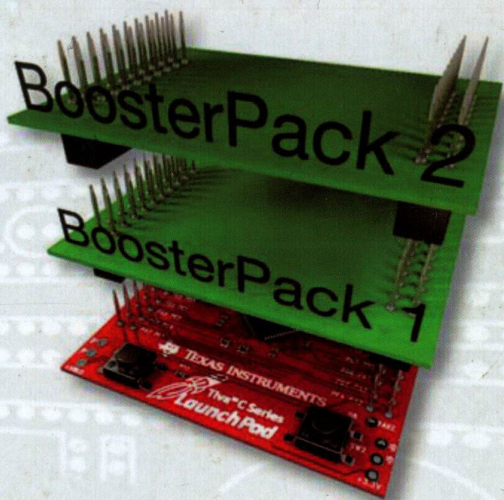
[11] TI Stellaris ARM Cortex－M4F Blizzard Peripheral 概述, 2011.

[12] Getting Started with the Tiva TM4C123G LaunchPad Workshop Student Guide and Lab Manual, 2013.

在线互动交流平台

官方微博：http://weibo.com/cmpjsj
豆 瓣 网：http://site.douban.com/139085/
读者信箱：cmp_itbook@163.com

基于固件的ARM Cortex M4原理及应用



内容提要

本书围绕TI TM4C123G的固件库函数这一主线，介绍了TM4C123G6HPM微处理器的基本外设特点、结构与功能，固件库的函数功能及其使用。本书采用了真实硬件EK-TM4C123GXL LaunchPad实验板（包括DK-TM4C123G）与虚拟硬件Proteus 8.1相结合的方式来介绍基于固件的软件编程与测试方法，以利于有真实板卡但资源不足或无EK-TM4C123GXL板卡的读者学习与测试基于固件的代码。

本书可供嵌入式工程师在基于固件的ARM Cortex M4开发时查阅，也可作为高校电类专业学习ARM Cortex M4的入门教材。

地址：北京市百万庄大街22号
邮政编码：100037
电话服务
服务咨询热线：010-88361066
读者购书热线：010-68326294
010-88379203
网络服务
机 工 官 网：www.cmpbook.com
机 工 官 博：weibo.com/cmp1952
教育服务网：www.cmpedu.com
金 书 网：www.golden-book.com
封面无防伪标均为盗版



机械工业出版社
微信公众号


计算机分社微信服务号

上架指导 嵌入式技术/ARM Cortex

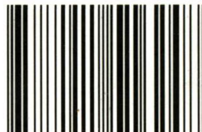
ISBN 978-7-111-51624-8

策划编辑◎ 时 静 / 封面设计◎



子时文化 ZiShi Culture

ISBN 978-7-111-51624-8



定价：89.90元